AFRL-IF-RS-TR-2003-174
Final Technical Report
July 2003

A CASE-BASED REFLECTIVE NEGOTIATION MODEL

University of Kansas

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. JH357

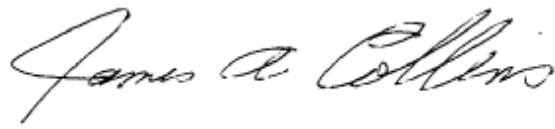*APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.*

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2003-174 has been reviewed and is approved for publication.

APPROVED:　　ROBERT J. PARAGI
　　　　　　　Project Engineer

FOR THE DIRECTOR:　JAMES A. COLLINS, Acting Chief
　　　　　　　　　　Information Technology Division
　　　　　　　　　　Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE JULY 2003	3. REPORT TYPE AND DATES COVERED Final Jun 99 – Apr 03

4. TITLE AND SUBTITLE
A CASE-BASED REFLECTIVE NEGOTIATION MODEL

5. FUNDING NUMBERS
C   - F30602-99-2-0502
PE - 62301E
PR - H357
TA - 01
WU - 01

6. AUTHOR(S)
Costas Tsatsoulis

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of Kansas
2383 Irving Hill Road
Lawrence Kansas 66044-7552

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Defense Advanced Research Projects Agency   AFRL/IFTB
3701 North Fairfax Drive   525 Brooks Road
Arlington Virginia 22203-1714   Rome New York 13441-4505

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

AFRL-IF-RS-TR-2003-174

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: Robert J. Paragi/IFTB/(315) 330-3547/ Robert.Paragi@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 Words)*
The goal of our work was to create autonomous agents that worked in a domain of constrained resources and that used case-based negotiation to allocate resources in a satisfying manner. The domain of application was distributed sensor management for multi-target tracking. We developed two separate but similar formalisms for the implementation of the negotiation agents: first, a real-time architecture based on the Belief-Desire-Intention (BDI) agent framework and second, a near real-time architecture that relied on domain heuristics. Both architectures used Case-Based Reasoning (CBR) to select a negotiation strategy, and then negotiated using argumentation. The BDI architecture used a real-time Linux kernel to have time awareness. In addition to this main thrust of our work, we were also tasked to provide tools for analyzing the performance of Java code, real-time Linux services, a Communication Server to model the RF communication channels of the sensors, and for studying and improving the Challenge Problem code.

14. SUBJECT TERMS
Autonomous Negotiation, Autonomous Agents, Case-Based Reasoning, Intelligent Agents

15. NUMBER OF PAGES
50

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102

Table of Contents

List of Illustrations

1. Introduction

The goal of our work was to create autonomous agents that worked in a domain of constrained resources and that used case-based negotiation to allocate resources in a good-enough soon-enough manner. The sponsor elected the domain of application to be distributed sensor management for multi-target tracking. We developed two separate but similar formalisms for the implementation of the negotiation agents: first, a real-time architecture based on the Belief-Desire-Intention (BDI) agent framework, and second, a near real-time architecture that relied on domain heuristics. Both architectures are described in the rest of this document.

In addition to this main thrust of our work, we were also tasked to provide tools for analyzing the performance of Java code, real-time Linux services, a Communication Server to model the RF communication channels of the sensors, and for studying and improving the Challenge Problem code.

2. Real-Time, Case-Based Negotiating Agents using BDI

While negotiation has been used in the past in problem solving in multiagent systems, in our work we concentrate on negotiations and activities that must occur in real time. The introduction of hard real time in negotiation and action execution complicates the problem greatly, and existing negotiation protocols cannot provide an adequate solution. In this report we describe a real-time negotiation model that is used in resource allocation problems. As an example domain we use multi-sensor target tracking, where each agent controls a sensor with a limited sensing coverage area. As a target moves across space, agents have to cooperate to track it. Each agent (together with the sensor it controls) consumes resources such as time, battery power, bandwidth of the communication channel, and some percentage of the CPU where the agent resides, and each agent strives to manage and utilize its resources efficiently and effectively. This motivates the agents to share their knowledge about a problem based on their viewpoints in their effort of arriving at a solution. The problem of global resource allocation becomes a problem of locally negotiated compromises and local constraint satisfaction.

We developed a logical negotiation protocol that incorporates a real-time BDI model (Rao & Georgeff, 1991, 1995) to dictate the rules of encounter among our autonomous agents. A feature of our problem involves generating a "good-enough, soon-enough" solution[1] to resource allocation. Since time is critical—for example, to make a good triangulation for the location of a target, three different sensors have to make a measurement within 2 seconds of each other—agents use time to guide their negotiation behavior. We base our temporal model on (Allen 1983) in which logical events or propositions can be ordered consistently along a timeline and durations of events or propositions holding true can be derived from their relationships with others. This temporal logic allows us to define explicitly the transition of a BDI state to another, including causality and co-existence. Equipped with the definition of time, we are able to model our negotiation activity with more accuracy, spelling out how and when a state changes and how and when it changes with other states, such as state s_1 triggers state s_2, s_1 has to occur before s_2, s_1 must hold true for some time during which s_2 must hold true as well, and so on. Therefore, states may change their truth values during a reasoning process as long as the states are needed to hold constant during that time period releasing other states to be updated or

---

[1] A "good-enough, soon-enough" solution is sometimes known as a "satisficing" one—a solution that satisfies the minimum problem requirements and the time constraints.

changed by other events or states. This is critical in our agent design as each agent is multi-threaded, meaning that several threads may attempt to access and modify the same variable (state) at the same time. To maintain data integrity, when a thread is accessing or modifying a variable, other threads will be blocked, and this is the common approach. However, software designers must find out how and for how long the threads will be blocked; and this information is very important in a real-time system like ours. With the temporal logic, we know for how long threads will be blocked and how these threads will be blocked awaiting which variables (states) to become accessible. This allows us to fine-tune the system to increase the efficiency of the negotiation process. Further, by incorporating temporal logic and BDI models into our negotiation protocol, we can guarantee both logically and temporally the completion of a negotiation. We know what states are needed (and when they are needed) for a negotiation to logically complete, and we also can model the time distribution or usage needed for each step of the negotiation to complete within certain time constraints. Note that temporal components have been in place in the BDI model (Rao and Georgeff 1991, Cohen and Levesque 1990) to determine how the three modalities are related over time. For example, taking time into consideration allows one to have persistent intentions, inevitable outcomes, and so on. In our model, we use the temporal logic to control the stability of a state, which in turn facilitates our multi-threaded solution.

We further define two sets of communicative acts—one for handling incoming messages and one for handling outgoing messages. A communicative act that handles incoming messages is a function that turns an event (the arrival of a message) to a set of BDI states. The function parses the incoming message and generates states that are necessary for the agent reasoning during a negotiation process. Similarly, a communicative act that handles outgoing messages is a function that composes a message based on the agent's current BDI states and sends it out via a communication channel. We qualify these acts with temporal logic and incorporate them into the negotiation protocol. In addition to the communicative acts, we utilize a suite of real-time enabling functional predicates to assist agents in negotiations. These predicates are events that take time to execute and they also generate or modify states.

Since each agent is autonomous and reacts to its environment, each has its own knowledge base and its own monitoring of the world events, including its sensor, its neighbors, and the targets. To increase the fault tolerance of the multi-agent system, each agent is responsible only for the resources it controls (in our example domain, its sensor and associated components), and it controls the minimal set of resources it requires to achieve its task. In our work agents have minimal knowledge and information—they know how to perform their tasks, have a local, limited view of the world provided to them by the equipment they control, and know of the existence of other similar agents, but they do not have an explicit view of the information of the other agents. There is some implicit knowledge, namely that the other agents control a set of resources, that they are willing to cooperate, that they are capable of negotiation for resource sharing, and that they are truthful. To establish a common reasoning basis during a collaborative effort, an agent is required to communicate to its potential partner why it needs to share the resources controlled by the partner. This knowledge exchange can be done via different mechanisms such as a blackboard where agents post information on a common site, or auctions where a contractor-agent oversees the message passing among contractee-agents, or through agent-based negotiations where agents exchange information directly. In our approach, we use negotiations motivated by a global goal—to track as many targets as accurately as possible—guided by a set of local optimization criteria that affect the strategies. During a negotiation

agents exchange information of their individual viewpoints of the current (and relevant) world situation. In this manner, the agents are able to argue and attempt to persuade each other explicitly, resulting in efficient knowledge transfer. We thus do away with a centralized information facility that requires constant updates and polling from agents, and, instead, knowledge is exchanged when necessary resulting in less communication traffic. Knowledge inconsistencies are resolved in a task-driven manner, making knowledge management easier.

One important part of the negotiation process is the determination of the negotiation strategy based on the current task description. To do so, we use a model derived from case-based reasoning (CBR) (Kolodner 1993) that is time-constrained and that retrieves the most similar cases, selects the best case based on utility theory, adapts the case to the current situation, and then uses the case's negotiation strategy to perform negotiations. The CBR approach limits the time needed to decide on a negotiation strategy—selection (through retrieval) and generation (through adaptation) of a situation-appropriate strategy—and enables the agent to learn autonomously and adapt itself to different scenarios in the domain.

2.1. Agent Characteristics

Each agent has the following characteristics:
(1) Autonomous – Each agent runs without interaction with human users. It maintains its own knowledge base, makes its own decisions, and interacts with its sensor, neighbors and environment.
(2) Rational – Each agent is rational in that it knows what its goals are and can reason and choose from a set of options and make an advantageous decision to achieve its goal (Wooldridge and Jennings 1995).
(3) Communicative – Each agent is able to communicate with others, by initiating and responding to messages, and carrying out conversations.
(4) Reflective (or Aware) – According to Brazier and Treur (1996), a reflective agent reasons based on its own observations, its own information state and assumptions, its communication with another agent and another agent's reasoning, and its own control or reasoning and actions. By being reflective, each agent is time aware and situationally aware. When an agent is time aware, it observes time in its decision making and actions. Its reasoning takes time into account, and thus, the outcome of a reasoning process is partially dependent on time. When an agent is situationally aware, it observes its current situation, the situation of its neighbors, and that of the world and makes decisions based on these observations. In general, an agent that is situationally aware observes the resources that it shares with other agents, its current tasks, messages, profiles and actions of its neighbors, and the external changes in the environment. In our work we require a stronger level of situational awareness. An agent also observes its own resources that *sustain the being* of the agent. For a hardware agent, these resources may be the battery power, the radio frequency links, etc. For a software agent, these resources may be CPU, RAM, disk space, communication channels, etc. Note that, for example, in (Sandholm & Lesser 1995), a *bounded rationality* model is used where each agent has to pay for the computational resources (CPU cycles) that it uses for deliberation, assuming that the resources are available. In our model, however, we require an agent to be aware of whether the resources are available before even starting a negotiation.
(5) Honest – Each agent does not knowingly lie or intentionally give false information. This characteristic is also known as veracity (Galliers 1988).

(6) Adaptive – Each agent is able to adapt to changes in the environment and learns to perform a task better, not only reactively but also from its past experience.
(7) Cooperative – Each agent is motivated to cooperate if possible with its neighbors to achieve global goals while satisfying local constraints.

Generally, the agents in a multi-agent system may be controlling different resources and use different reasoning and negotiation techniques. In our approach, we require (1) that all agents be capable of negotiation in which they share a common vocabulary that enables message understanding, and (2) that each agent knows what resources may be used or controlled by a non-empty subset of the other agents in the environment so that it can determine whom to negotiate with. In our particular domain of application, each agent controls the same resources, since each one controls the same type of sensor. Also, each agent uses the same negotiation methodology based on case-based reasoning, but the individual case bases differ.

Formally, our multi-agent system architecture is defined as follows. Suppose that we denote a multi-agent system as Ω. Suppose that we define a neighborhood of an agent α_i, Ψ_{α_i}, such that $\Psi_{\alpha_i} \subseteq \Omega$, $\Psi_{\alpha_i} \neq \varnothing$, and that the agent α_i knows about all other agents in the neighborhood. Thus, we have $\alpha_i, \alpha_j \in \Psi_{\alpha_i}, \forall j\, \lambda(\alpha_i, \alpha_j)$ where $\lambda(a,b)$ means agent a knows about the existence of agent b and can communicate with agent b. A neighborhood is different from a team as defined in (Tambe, 1997), which is task-driven and formed among a set of agents to accomplish a task. A neighborhood is a subset of agents of the multiagent system that *could* form a team. In our particular domain of application (multisensor target tracking), a neighborhood consists of a set of agents that control sensors that are physically close and whose sensing beams overlap. So, in our multi-agent system Ω, there is a set of neighborhoods, $\Omega = \{\Psi_{\alpha_1}, \Psi_{\alpha_2}, ..., \Psi_{\alpha_N}\}$, and each neighborhood can form any number of teams. Neighborhoods do not necessarily have the same number of members, and neighborhoods may share members.

When a target is sensed, an agent tracks the target, refers to its neighborhood information, and dynamically forms a *negotiation coalition*, that is, a subgroup of its neighborhood agents with which it *may* negotiate to request resources to assist it in its task. For example, when an agent detects a target in its sensing area, the agent immediately obtains an estimate on the position and velocity of the target. It then projects the future positions of the target and identifies the neighbors whose sensors are able to cover the target moving in the projected path. These are agents that control resources (i.e. sensor beams) that it needs to track the target, and these are the agents that will be part of the negotiation coalition.

2.2. A Logical Protocol for Real-Time Argumentative Agent Negotiations

In this section, we describe the logical protocol for our real-time argumentative agent negotiations. Argumentative negotiations differ from traditional negotiations because the agents conducting the former negotiate about *why* one of the agents needs to perform a certain task in addition to what the task is. Therefore, our work is similar to (Parsons *et al.* 1998). However, we assume that agents have the same inference rules. Parsons' work assumes that agents may have different ones, and thus his argumentation protocol requires agents to exchange inference rules as well. Moreover, we incorporate real-time issues into our design guidelines.

Figure 1 shows our negotiation protocol in a state diagram between two agents: *a* and *b*. State 0 is the initial state, the double-circle. State 1 is the first *handshake* state, indicating whether the initiated negotiation will be entertained. State 4 is the initiating state while state 5 is the responding state. The initiating state is where the initiating agent, *a*, returns to, basically the processing loop of the negotiator module. The responding state is where the responding agent, *b*, returns to, respectively. Agent *a* initiates a negotiation request to *b* by sending an INITIATE message (initiate(*a*,*b*)), the state transitions to 1. At this juncture, there are four possible scenarios. First, agent *b* may outright refuse to negotiate by sending a NO_GO message (no_go(*b*,*a*)). This results in a final state of failure (state 2, rejected). Second, agent *b* may outright agree to the requested task by sending an AGREE message (agree(*b*,*a*)). This results in a final state of success (state 3). Third, agent *b* may decide to entertain the negotiation request and thus sends back a RESPOND message (respond(*b*,*a*)). This transitions the state to 4. Fourth, there may be no response from agent *b*. Thus agent *a*, after waiting for some time, has no choice but to declare a no response (no_response(*a*)) and moves to a state of failure (state 8, channel _jammed).

When the agents move to state 4, the argumentative negotiation begins and iterates between states 4 and 5 until one side opts out or both sides opt out or both sides agree. During the negotiation, (1) agent *a* provides information or arguments to *b* by sending INFO messages (info(*a*,*b*)), (2) agent *b* demands information or arguments from *a* by sending MORE_INFO messages (more_info(*b*,*a*)), (3) if agent *a* runs out of arguments, it sends a INFO_NULL message to *b* (info_null(*a*,*b*)), (4) if agent *b* runs out of patience, it counter-proposes by sending a COUNTER message to *a* (counter(*b*,*a*)), and (5) agent *a* can agree to the counter offer (agree(*a*,*b*)) and move to the state of success (state 3), or provide more information (info(*a*,*b*)) as requested, or provide no information (info_null(*a*,*b*)) if it has time to do so, hoping that agent *b* might come up with a better offer, or simply disagrees (abort(*a*,*b*)). Thus, an initiating agent will always negotiate until it has run out of time or when the responding agent opts out. However, an initiating agent may abort a negotiation, and this is where the conditions come into play. If the agent realizes that it has already obtained what it wants from other negotiations happening in parallel, then it aborts the current negotiation; or if the agent realizes that it no longer cares about the current negotiation, then it aborts. These conditions are based on desires and intentions, which in turns are based on beliefs of the agent. When an agent runs out of time, it issues an OUT_OF_TIME message to the other agent and quits the negotiation with a failure (state 6, out_of_time). When an agent aborts, it issues an ABORT message to the other agent and quits the negotiation with a failure (state 7, abort). Finally, whenever an agent does not hear from the other agent within an allocated time period, it assumes that the communication channel has been jammed or congested and quits with a failure (state 8, channel_jammed). Note that we distinguish NO_GO, STOP, OUT_OF_TIME, and ABORT in the above protocol. With the above different end states, agent *a* can determine whether the negotiation has failed because it has exhausted all its arguments (STOP) or otherwise and subsequently learn from the failure.
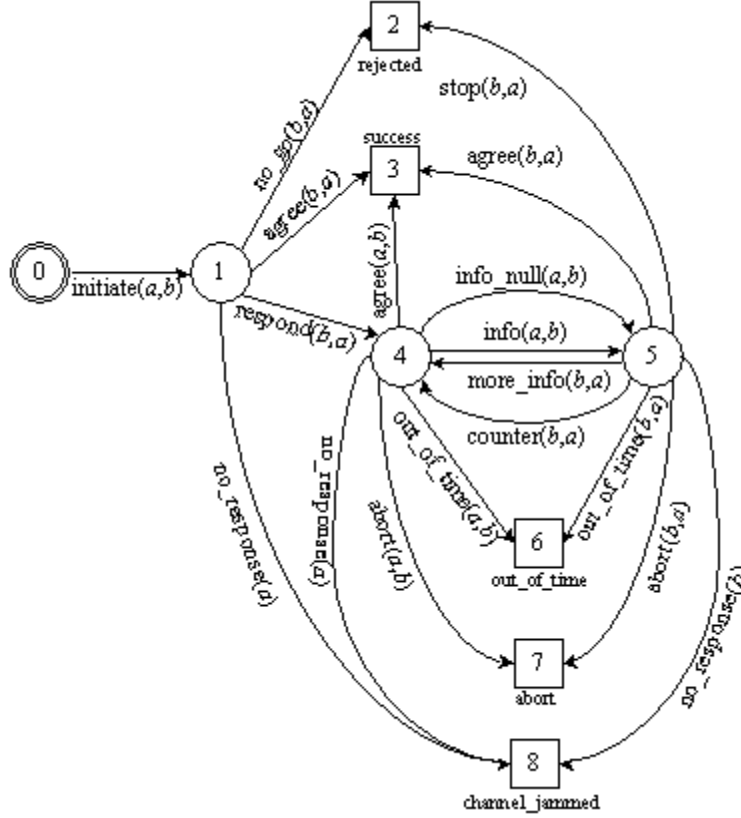
Figure 1 Our negotiation protocol. Squares are final states. The double-circle is the initial state.

We use an expanded multi-context BDI agents framework of Parsons *et al.* (1998), Noriega and Sierra (1996) to describe the logical framework of our negotiation protocol. As presented in Rao and Georgeff (1995), there are three modalities. First, beliefs (*B*) represent the states of the environment and the agent. There are also belief states that arise during negotiations, as agents learn each other's beliefs and intentions. Second, desires (*D*) represent the motivations of the agent. Third, intentions (*I*) represent the goals that the agent wants to achieve. We also assume the following axioms (Parsons *et al.* 1998):

1. $B : B(p \rightarrow q) \rightarrow (B(p) \rightarrow B(q))$
2. $B : B(p) \rightarrow \neg B(\neg p)$
3. $B : B(p) \rightarrow B(B(p))$
4. $B : \neg B(p) \rightarrow B(\neg B(p))$

Axiom 1 states that if one believes that *p* implies *q*, then if one believes *p*, then one believes *q* as a consequence. Axiom 2 states that if one believes *p*, then one does not believe the negation of *p*. Axiom 3 states that if one believes *p*, then one also believes that it believes *p*. This ensures that an agent *knows* and believes in what it believes. Similarly, Axiom 4 states that if one does not believe *p*, then one believes that it does not believe *p*. Similar axioms for desires and intentions are:

5. $D : D(p \rightarrow q) \rightarrow (D(p) \rightarrow D(q))$
6. $D : D(p) \rightarrow \neg D(\neg p)$

6

7. $I : I(p \rightarrow q) \rightarrow (I(p) \rightarrow I(q))$

8. $I : I(p) \rightarrow \neg I(\neg p)$

We adopt the strong realist BDI agent model of Rao and Georgeff (1991) such that (1) an agent only intends to do what it desires, and (2) an agent only desires what it believes. However, we do not adopt other rules presented in (Parsons *et al.* 1998), regarding the communication unit, because in our case (1) an intention for performing a task does not necessarily imply a communication of the performance of the task—in our domain, a task may sometimes be performed by the agent itself, and (2) we do not require an agent to report a completion of a task to another agent—because in our current design, an agent assumes that if another agent agrees to perform a task, it knows that that agent will try its best to perform and complete the task, and that whether it believes the task has been performed is no longer important. In the future, however, we plan to include a monitoring mechanism in an agent to enrich our real-time modeling of events and agent behavior in making better decisions. At that time, an agent would have to care whether a task has been performed successfully to update its own belief states, but would still not depend on the communicated information for the update.

We incorporate temporal logic into our protocol to explicitly define the various belief, desire, and intention states of an agent and when they are true. This is key to the real-time implementation of the protocol. To satisfy real-time constraints, our agent, as shown in Figure 1, consists of multiple concurrent processing threads. Each thread carries out a set of tasks, and these tasks access and modify the same states at different times. Some states must not be modified before certain actions have been carried out; some states should not be accessed before certain results have been obtained. To implement such a *state synchronization* across multiple concurrent processes, we use temporal logic to define when a state must be true and the duration for that state to stay true. Without the temporal logic component, it would have been close to intractable to manage the inter-thread, real-time activities.

As previously mentioned, Cohen and Levesque (1990) and Rao and Georgeff (1991) have incorporated temporal components into the BDI model. In Cohen and Levesque (1990), intentions are defined in terms of temporal sequences of an agent's beliefs and goals. Each possible world extendable from a current state at a particular time point is a time line representing a sequence of events. As such, the inter-modal relationships are stronger than those in Rao and Georgeff (1991). For example, an agent fanatically committed to its intentions will maintain its goals until either they are believed to be achieved or believed to be unachievable. Thus, intentions are seen as a *special* class of desires. Rao and Georgeff (1991), on the other hand, present an alternative possible-worlds formalism for BDI-architectures. Instead of a time line, they choose to model the world using a temporal structure with a branching time future and a single past, called a time tree, where a particular time point in a particular world is called a situation. There are three crucial elements to the formalism. First, intentions are on a par with beliefs and goals. This allows them to define different strategies of commitment and to model a wide variety of agents such as blinded, single-minded, and open-minded agents. Second, they distinguish between the choice an agent has over the actions it can perform and the possibilities of different outcomes of an action, factoring in the uncertainty that the environment brings into the determination of the outcomes. Third, they specify an interrelationship between beliefs, goals, and intentions that allows them to avoid problems such as commitment to unwanted side effects.

Compared to our proposed model below, the incorporation of the temporal elements is different. In Cohen and Levesque (1990), the temporal component was used to define intentions through commitment and persistence, derived from beliefs and goals. In Rao and Georgeff (1991), it was used to order possible worlds from a situation in both the time and space dimensions. Each time tree denotes the optional courses of events choosable by an agent in a particular world. For example, an agent has a belief ϕ, denoted $B(\phi)$, at time point t if and only if ϕ is true in all the belief-accessible worlds of the agent at time t. In our model, however, we use the temporal component to define the temporal duration that a state needs to be stable, or needs to occur in order for another state to take place. Thus, our motivation is to help design and implement agents that are multi-threaded and multi-tasking. We do not use temporal logic to define the BDI modalities. In the following subsections, we formalize our theory of the actions depicted in the state diagram of Figure 1.

To incorporate real-time concerns into our logical negotiation protocol, we use several interval relationships outlined in Allen (1983) and Allen and Ferguson (1994). Each interval t has a start time, t_s, and a finish time, t_f, and its duration is $t_f - t_s$. If $t_f - t_s$ equals the smallest amount within the resolution of the domain problem, then the interval becomes a *moment* or a *point*. There are seven basic relations between temporal intervals:
1. *Before*(i, j) where interval i ends before interval j.
2. *Meets*(i, j) where as soon as i finishes, interval j starts, i.e., the two intervals are consecutive.
3. *Overlaps*(i, j) where a portion of interval i overlaps a portion of interval j in time and i starts before j and i ends before j.
4. *Starts*(i, j) where interval i starts at the same time as interval j but interval i has a shorter duration.
5. *Finishes*(i, j) where interval i finishes at the same time as interval j but interval i has a shorter duration.
6. *During*(i, j) where interval i starts after interval j and interval i ends before interval j.
7. *Equals*(i, j) where both intervals have the same durations and start and end at the same times.

We also adopt the homogeneity axiom schema such that a proposition is homogeneous if and only if when it holds over an interval t, it also holds over any sub-interval within t. Within the framework of our negotiation protocol, the BDI states are all homogeneous propositions, hence the use of strong negation $\neg$. The predicates such as the communicative acts are anti-homogeneous (Allen 1991) since, for example, the action of composing and sending a message is a process that does not generally hold unless completed in the end. We also introduce a notational convenience $[e]$ as the interval of an event/action, or, in the case of homogeneous positions, as the interval of a proposition holding true.

In our protocol, we have two sets of corresponding communicative acts. One is for receiving and parsing an incoming message; the other for composing and sending an outgoing message. For example, a communicative act that composes a negotiation request and initiates contact with a potential negotiation partner is of the following form:

$$C_{out} : initiate(i, r, Do(r, \rho), t)$$

where the predicate *initiate* is the communicative act (composing and sending), *i* is the initiating agent, *r* is the responding agent, $Do(r,\rho)$ is the requested task, i.e., "*r* do task ρ," and *t* is the time taken for the communicative act to start and finish.

A communicative act is an event that performs a set of tasks consecutively such that the sum of the durations of the tasks is the duration of the communicative act. A communicative act may generate new propositions, may cause a new state externally to another agent, and may be terminated by another proposition or event. For example, at the end of the interval *t*, if the communication is successful, then the responding agent will receive the request, $C_{in} : initiate(i,r,Do(r,\rho),k)$, where, in theory, $Meets(t,k)$, and in practice (due to communication latency), $Before(t,k)$. On the other hand, if the communication is unsuccessful (e.g., due to communication channel being jammed), then the predicate *initiate* of the initiating agent will be terminated by a terminator (Vere 1983). To simplify our discussions, we use $C_{in} : f(\langle sender \rangle, \langle receiver \rangle, \langle request \rangle, t)$ and $C_{out} : f(\langle sender \rangle, \langle receiver \rangle, \langle request \rangle, t)$ to differentiate between incoming and outgoing message handling, where *f* is one of the acts defined in our protocol. Currently, we have the following communicative acts, as depicted in Figure 2: *initiate*, *respond*, *no_go*, *agree*, *abort*, *out_of_time*, *counter*, *more_info*, *info*, *info_null*, and *stop*, for a total of 22.

One of the objects generated by a C_{out} communicative act is the message. Our message syntax is $msg(\langle sender \rangle, \langle receiver \rangle, \langle type \rangle, \langle request \rangle, \langle contents \rangle)$ where the type of the message denotes one of the communicative acts and the contents consist of whatever pertinent to the request. In the following discussion, we often mention the messages in the same breath as the communicative acts and use them interchangeably.

Vere (1983) described a proposition as bounded by its holding true over a time interval, with a limited life span terminated by later contradictory assertions. Under Vere's definition, an assertion *T* is a terminator for an assertion *A* if and only if: (1) *A* and *T* are contradictory, (2) *T* follows *A* in time, and (3) no assertion *T′* exists satisfying the first two conditions such that *T* follows *T′* in time. Basically, $Meets([A],[T])$.

Take our communicative acts for example. Suppose an initiating agent performs *initiate*. As we shall see later, the agent has a set of BDI states (including its belief that the communication channel is operating, $B : B_i(channel\_good,t)$) that holds true such that the collective interval, Θ, of those states overlaps the communicative act's interval, $Overlaps([\Theta],[initiate])$. That is, the communicative act may continue after Θ no longer holds. Further, if the communication fails, which in this case means the message is not sent, the agent generates a belief $B : \neg B_i(channel\_good,k)$. This becomes a terminator of *initiate* since one of the preconditions for the communicative act and the new belief state are contradictory and $Meets(t,k)$. As a result, the action is terminated. Note that a proposition or an assertion in our logical negotiation protocol holds until terminated by a terminator, and so does a terminator. A termination may lead to the stoppage of a set of actions or tasks, which in turn may lead to the stoppage of a set of events. An action or task may, however, terminate by itself normally as it completes within a time interval.

2.3. Coalition Formation

When an agent, for example, senses a target, it needs to form a coalition from which to ask for help. To do so, it collects all its current BDI states and external information to obtain a list of potentially helpful neighbors. Since each task is new, that means in the beginning of the coalition formation, there exist no belief states such that $B : B_\alpha(CanDo(n, \rho), t_B)$ where n is a member of the set of all known neighbors, N_α, of the agent α and ρ is the new task. After a domain-specific search process, the agent obtains a list of potentially helpful neighbors, $N'_\alpha \subseteq N_\alpha$. At this point, equipped with this list, the agent $\forall n \; B : B_\alpha(CanDo(n, \rho), t_B)$ where $n \in N'_\alpha$. Then the agent checks its needs and creates desires for only a certain number of these neighbors to perform the tasks (since enlisting everybody one knows to help out a task is counter-productive). In our actual agent design, these neighbors are ranked according to their utility values. The agent then checks the number of available negotiation threads that it may use to negotiate with these neighbors. That number trims N'_α to obtain N''_α. At this point, the agent $\forall n \; D : D_\alpha(Do(n, \rho), t_D)$ where $n \in N''_\alpha$. Subsequently, as the agent begins to send out requests, with each successful contact, the agent forms $I : I_\alpha(Negotiate(n, Do(n, \rho)), t_I)$ with the neighbor n that has made contact. In the end, all neighbors contacted are in the coalition $C_\alpha(\vec{\rho})$ such that $C_\alpha(\vec{\rho}) \subseteq N''_\alpha$ and where $\vec{\rho} = \{\rho_1, \rho_2, \cdots, \rho_{|C_\alpha(\vec{\rho})|}\}$ is a set of subtasks that contribute to the original task. In this way, each neighbor is contacted to perform a subtask and a coalition formation drives an agent to negotiate with its neighbors. In terms of the temporal interval relationships, since a negotiation process is stepwise and interruptible, if an agent does not have the desire for a neighbor to perform a task, then it does not intend to negotiate with that neighbor regarding that particular task. Thus, we have the following condition: $During(t_D, t_B) \wedge During(t_I, t_D)$.

Note that in our model, we have N 1-to-1 negotiations but do not conduct *direct* 1-to-*N* negotiations. That is, during a negotiation, a negotiation thread does not consult directly other negotiation threads of the same agent. However, the parent agent of the negotiation threads does examine the completion status of its negotiation threads and may change the beliefs, desires, and intentions of the negotiation threads through negotiation-related predicates due to the results of other negotiations. This design choice is motivated by real-time concerns. Instead of having the negotiator module of a negotiation thread monitoring the activities of other negotiation threads of the same agent, the core thread of the agent monitors the negotiation activities through the coalition manager. The coalition manager determines whether a coalition is still viable, whether a coalition has been achieved, and whether a coalition is to be aborted, and commands each individual negotiation thread accordingly. Thus, each negotiator can concentrate on their negotiation task at hand.

Our coalition formation consists of three stages: (1) coalition initialization where an agent obtains a ranked list of potentially helpful neighbors, $N'_\alpha \subseteq N_\alpha$ based on the current problem, (2) coalition finalization where the agent contacts the neighbors in $N'_\alpha \subseteq N_\alpha$ to negotiate, and (3) coalition acknowledgment where the agent concludes the success or failure of the coalition and inform neighbors who have agreed to help. In coalition initialization, a neighbor is ranked based on its potential utility in helping with the current problem. This potential utility is based on the past and current relationships between the agent and the neighbor, and the ability of the neighbor with the current problem. For example, if the target is moving towards the sensor coverage of the neighbor and will be inside the coverage for a long time, then the neighbor has a high

potential utility. During the coalition finalization step, the agent negotiates with the ranked neighbors concurrently. As each negotiation thread reports its final status the core thread (the parent agent), the parent agent decides whether to abort meaningless negotiations or to modify negotiation tactics. After the finalization step, the agent knows whether it has a coalition. If it does, it sends a confirmation message to all the neighbors who have agreed to help. If it does not, it sends a discard message to those who have agreed to help; this is the acknowledgment step.

2.4. Complete Negotiation Rules and Protocol

Our agents are cooperative and are also directed to satisfy global goals. Each is motivated to look for help from its neighbors and to entertain negotiation requests from its neighbors, and each genuinely wishes to have a successful negotiation. First, we have the implicit assumption $D : D_\alpha(Cooperate, t_{always})$ where $t_{always,s} = time0$ is the start time and $t_{always,f} = \infty$ is the finish time, meaning that the desire is always true. Second, we have $D : D_\alpha(Satisfy - Global - Goal, t_{always})$.

When an initiating agent, i, negotiates with a neighbor, r, the agent has the following BDI states, as shown in the previous section:

$$D : D_i(Cooperate, t_{always}) \wedge B : B_i(CanDo(r, \rho), t_B) \wedge$$
$$D : D_i(Do(r, \rho), t_D) \wedge I : I_i(Negotiate(r, Do(r, \rho)), t_I).$$

Combining the above states with $D : D_i(Satisfy - Global - Goal, t_{always})$, we assume that each agent:

$$I : I_i(succeed(Negotiate(r, Do(r, \rho))), t_I).$$

The above intention motivates an initiating agent to continue negotiating. In the later discussion, we use this intention to explicitly drive the negotiation axioms, while keeping other BDI states implicit.

When a responding agent, r, receives a request to negotiate from an initiating agent, i, the agent parses the message and examines its own current states. If it believes it can perform the requested task yet does not have the desire to do so, then because of the desire to cooperate, it has the following BDI states:

$$D : D_r(Cooperate, t_{always}) \wedge B : B_r(CanDo(r, \rho), t_B) \wedge$$
$$\neg \exists D : D_r(Do(r, \rho), t_D) \wedge I : I_r(Negotiate(i, Do(r, \rho)), t_I).$$

Similarly, combining the above states with $D : D_r(Satisfy - Global - Goal, t_{always})$, we have the following that keeps the agent negotiating:

$$I : I_r(succeed(Negotiate(i, Do(r, \rho))), t_I).$$

A completely successful negotiation results in $D : D_r(Do(r, \rho), t_D)$. The negotiating strategy of an initiating agent is to help the responding agent achieve $D : D_r(Do(r, \rho), t_D)$ while that of the

responding agent is to let itself be persuaded by the initiating agent's arguments in order to achieve $D : D_r(Do(r, \rho), t_D)$. As we will see in the next two sections, there are partially successful negotiations and different types of failures. Furthermore, we have to deal with the duration of a BDI state. A BDI state holds true only sufficiently long: (1) no longer than the duration of the task to be performed, or (2) until ended by a terminator. For the implicit assumptions, the tasks *Cooperate* and *Satisfy-Global-Goal* have infinite duration. So, in $D : D_r(Do(r, \rho), t_D)$, the agent r only desires to do the task for at most the duration of the task ρ. Thus, we have $During(t_D, [Do(r, \rho)])$ where $[Do(r, \rho)]$ is the time interval for r performing the task. This assumption applies to all BDI states related to performing a task. On the other hand, when a negation of performing a task is involved, such as $D : D_r(\neg Do(r, \rho), t_D)$, the agent cannot rely on $[Do(r, \rho)]$ to quantify t_D. In our design, instead of making t_D a *moment* or a *point*, we let it be until terminated by another assertion/proposition (e.g., generated by a later coalition process). These assumptions allow an agent to negotiate regarding the same task at two different times as long as the task negotiated first has completed since the BDI states of performing a task are self-terminating and those of *not* performing a task are terminated by assertions generated by other agent activities.

Finally, we have to deal with arguments since our negotiation approach is argumentation-based. Suppose the request is for the responding agent r to perform the task $\rho : Do(r, \rho)$. The responding agent has a set of internal arguments, Γ_r, for and against performing the task. If the agent believes it can perform the task but $\Gamma_r \nvDash D_r(Do(r, \rho), t_D)$ (meaning the arguments do not support the desire for performing the task), then it has to rely on the initiating agent for more arguments. The initiating agent has its own set of arguments, Γ_i, for the responding agent performing the task. The underlying approach is to send over a subset Γ_i' of Γ_i to the responding agent until

$$\Gamma_r \cup \Gamma_i' \vDash D_r(Do(r, \rho), t_D) \text{ (in which case the negotiation succeeds)},$$

or until

$$\Gamma_r \cup \Gamma_i \nvDash D_r(Do(r, \rho), t_D) \text{ (in which case the negotiation fails)},$$

where $\Gamma_i' \subseteq \Gamma_i$ is the set of arguments already communicated to the responding agent from the initiating agent. This assumption is a critical element in our negotiation protocol as it facilitates a stepwise evaluation of arguments to move closer to a conclusion of the negotiation.

2.4.1. Initiating Behavior

Here we outline the axioms that link an agent's communication and its internal states for conducting negotiations as an initiating agent.

Initiate. When an initiating agent (*i*) believes that it intends to negotiate with the responding agent (*r*) to perform a task ρ, it initiates a negotiation request to the responding agent.

$$I : I_i\big(Negotiate(r, Do(r, \rho)), t_I\big) \Rightarrow C_{out} : initiate\big(i, r, Do(r, \rho), t_{cout,initiate}\big)$$

where $During(t_{cout,initiate}, t_I)$. The predicate *initiate* encapsulates the act of composing an INITIATE-type message and sending the message to agent r.

Failure 1. When an initiating agent (i) receives a NO_GO message from a responding agent (r), it believes that the responding agent r cannot perform the requested task ρ and stops intending r to perform the task.

$$C_{in} : no\_go\big(r, i, Do(r, \rho), t_{cin,no\_go}\big) \wedge I : I_i\big(Negotiate(r, Do(r, \rho)), t_I\big) \Rightarrow$$
$$I : I_i\big(\neg Negotiate(r, Do(r, \rho)), t'_I\big) \wedge B : B_i\big(\neg CanDo(r, \rho), t_B\big) \wedge$$
$$D : D_i\big(\neg Do(r, \rho), t_D\big) \wedge rejected$$

where $During(t_{cin,no\_go}, t_I) \wedge Meets(t_I, t'_I) \wedge Meets(t_I, t_B) \wedge Starts(t_B, t_D)$. The *no_go* communicative act is the encapsulation of receiving and parsing a NO_GO (an outright refusal to negotiate) message. This rule allows an agent to move on to the next neighbor after the responding agent has outright refused to negotiate. This is real-time motivated: instead of trying to come up with a lesser task and trying to establish a negotiation with the responding agent, the initiating agent simply gives up and shifts its focus to other neighbors. The proposition *rejected* indicates the failure of a negotiation.

Note also that in the above rule, the changes in the internal states of the agent are triggered by the incoming message. This is one of our design characteristics and goals: agents communicate only when necessary since the environment is real-time and resource constrained, and information is exchanged only during negotiation.

Success 1. When an initiating agent (i) receives an AGREE message from a responding agent (r), it believes that the responding agent r intends to perform and will perform the requested task ρ.

$$C_{in} : agree\big(r, i, Do(r, \rho), t_{cin,agree}\big) \wedge I : I_i\big(Negotiate(r, Do(r, \rho)), t_I\big) \Rightarrow$$
$$B : B_i\big(D_r(Do(r, \rho)), t_B\big) \wedge I : I_i\big(\neg Negotiate(r, Do(r, \rho)), t'_I\big) \wedge D : D_i\big(\neg Do(r, \rho), t_D\big) \wedge success$$

where $During(t_{cin,agree}, t_I) \wedge Meets(t_I, t'_I) \wedge Meets(t_I, t_B) \wedge Starts(t_B, t_D)$. The *agree* communicative act is the encapsulation of receiving and parsing an AGREE message. In this axiom, if an agent receives an AGREE message from the responding agent, then (1) it believes that the responding agent desires to perform the task, (2) it intends no longer to negotiate, and (3) it desires no longer that the responding agent perform the task. This third desire may seem counter-intuitive at first glance. Its purpose is to say "If I believe that you have the desire to do the task, then I don't have to desire you to do the task anymore," and that does not prevent the agent to desire the responding agent to perform the task in the future. The proposition *success* indicates the success of a negotiation.

Info 1. When an initiating agent (i) receives a RESPOND message from a responding agent (r), it (1) believes that the responding agent r intends to negotiate and (2) intends to obtain a

successful negotiation. Consequently, the initiating agent *i* intends to help *r* to desire to perform the task by supplying available necessary information.

$$C_{in} : respond(r, i, Do(r, \rho), t_{cin,respond}) \wedge I : I_i(Negotiate(r, Do(r, \rho)), t_I) \wedge B : B_i(I_r(Negotiate(i, Do(r, \rho))), t_B)$$
$$\wedge I : I_i(succeed(Negotiate(r, Do(r, \rho))), t_I') \wedge \exists p : (p \in \Gamma_i \wedge p \notin \Gamma_i') \Rightarrow C_{out} : info(i, r, Do(r, \rho), t_{cout,info})$$

where

$$During(t_{cin,respond}, t_I) \wedge Finishes(t_I', t_I) \wedge Equals(t_B, t_I') \wedge Starts(t_{cout,info}, t_I') \wedge Before(t_{cin,respond}, t_{cout,info}).$$

The communicative act *respond* is the encapsulation of receiving and parsing a RESPOND message, and the communicative act *info* is the encapsulation of composing and sending an INFO message, including selecting a *p* from the set of arguments, Γ_i, that is not a member the set of arguments already sent, Γ_i' to *r*. Clause 1 of the axiom explicitly derives the motivation for the initiating agent to continue negotiating as it intends to have a successful negotiation since it now believes that the responding agent intends to negotiate as well. Clause 2 of the axiom drives the agent to send over more arguments to help bring a successful conclusion to the negotiation.

Info_null 1. When an initiating agent (*i*) receives a RESPOND message from a responding agent (*r*), it (1) believes that the responding agent *r* intends to negotiate and (2) intends to obtain a successful negotiation. However, if *i* has run out of information or arguments, then it notifies *r* that it can no longer provide arguments.

$$C_{in} : respond(r, i, Do(r, \rho), t_{cin,respond}) \wedge I : I_i(Negotiate(r, Do(r, \rho)), t_I) \wedge B : B_i(I_r(Negotiate(i, Do(r, \rho))), t_B)$$
$$\wedge I : I_i(succeed(Negotiate(r, Do(r, \rho))), t_I') \wedge \exists p : (p \in \Gamma_i \wedge p \notin \Gamma_i') \Rightarrow C_{out} : info\_null(i, r, Do(r, \rho), t_{cout,info\_null})$$

where

$$During(t_{cin,respond}, t_I) \wedge Finishes(t_I', t_I) \wedge Equals(t_B, t_I') \wedge Starts(t_{cout,info\_null}, t_I') \wedge Before(t_{cin,respond}, t_{cout,info\_null}).$$

This rule is the counterpart to *Info 1* previously discussed. When an agent runs out of arguments, it notifies the responding agent about it. Instead of giving up on the negotiation right away—since obviously the initiating agent knows that it has not been able to persuade the responding agent and now it has run out of arguments, the initiating agent informs the responding agent of its situation and hopefully the responding agent will be able to counter-offer. So, in a way, this shifts the responsibility of achieving a successful negotiation to the responding agent from the initiating agent. Up until this point, the initiating agent has been responsible for keeping the negotiation going by supplying arguments/information to the responding agent, trying to convince it. Finally when the initiating agent can argue no further, the decision shifts to the responding agent.

Info 2. When an initiating agent (*i*) receives a MORE_INFO message from a responding agent (*r*), it simply supplies more unused arguments.

$$C_{in} : more\_info(r, i, Do(r, \rho), t_{cin,more\_info}) \wedge I : I_i(succeed(Negotiate(r, Do(r, \rho))), t'_I)$$
$$\wedge \exists p : (p \in \Gamma \wedge p \notin \Gamma'_i) \Rightarrow C_{out} : info(i, r, Do(r, \rho), t_{cout,info})$$

where $During(t_{cin,more\_info}, t'_I) \wedge During(t_{cout,info}, t'_I) \wedge Before(t_{cin,more\_info}, t_{cout,info})$. This axiom is similar to *Info 1*.

Info_null 2. When an initiating agent (*i*) receives a MORE_INFO message from a responding agent (*r*), if it does not have any more arguments, then it notifies *r* of its status.

$$C_{in} : more\_info(r, i, Do(r, \rho), t_{cin,more\_info}) \wedge I : I_i(succeed(Negotiate(r, Do(r, \rho))), t'_I)$$
$$\wedge \neg \exists p : (p \in \Gamma_i \wedge p \notin \Gamma'_i) \Rightarrow C_{out} : info\_null(i, r, Do(r, \rho), t_{cout,info\_null})$$

where $During(t_{cin,more\_info}, t'_I) \wedge During(t_{cout,info\_null}, t'_I) \wedge Before(t_{cin,more\_info}, t_{cout,info\_null})$. This axiom is similar to *Info_null 1*.

Info 3. When an initiating agent (*i*) receives a counter-offer (ρ') from a responding agent (*r*), *i* believes that *r* desires to perform ρ'. However, if ρ' is not acceptable, then the agent *i* continues to send unused arguments to *r*.

$$C_{in} : counter(r, i, Do(r, \rho), t_{cin,counter}) \wedge B : B_i(D_r(Do(r, \rho')), t_B) \wedge B : B_i(\neg acceptable(\rho'), t'_B)$$
$$\wedge \exists p : (p \in \Gamma_i \wedge p \notin \Gamma'_i) \wedge I : I_i(succeed(Negotiate(r, Do(r, \rho))), t'_I) \Rightarrow C_{out} : info(i, r, Do(r, \rho), t_{cout,info})$$

where

$$During(t_{cin,counter}, t'_I) \wedge Meets(t_{cin,counter}, t_B) \wedge During(t'_B, t'_I)$$
$$\wedge Finishes(t'_B, t_B) \wedge Meets(t_B, t_{cout,info}) \wedge During(t_{cout,info}, t'_I).$$

This axiom is similar to *Info 1*. The communicative act *counter* is the encapsulation of receiving and parsing a COUNTER-type message, in which $\langle contents \rangle$ holds the counter-offer ρ'. Clause 1 states that when an initiating agent receives a counter-offer from the responding agent, it believes that the responding agent desires to perform the counter-offer. Now, the initiating agent checks the acceptability of the counter-offer. If the counter-offer is not acceptable and the agent still has unused arguments, then it sends over more arguments. This is how an initiating agent counter-offers a counter-offer: sending over more arguments in hope that the responding agent will come back with a better counter-offer, closer to the original request.

Note also the temporal relationships among $t_{cout,info}$, t_B, and t'_B. As soon as the initiating agent realizes that the counter-offer is not acceptable, both its beliefs that the responding agent desires to perform the counter-offer and that the counter-offer is unacceptable terminate and trigger the communicative act *info*. In other words, when the initiating agent counters a counter-offer, all beliefs regarding the counter-offer no longer hold.

Info_null 3. When an initiating agent (*i*) receives a counter-offer (ρ') from a responding agent (*r*), *i* believes that *r* desires to perform ρ'. However, if ρ' is not acceptable and the

agent does not have any more unused arguments, it notifies r that it can no longer provide arguments.

$$C_{in} : counter(r, i, Do(r, \rho), t_{cin,counter}) \land B : B_i(D_r(Do(r, \rho')), t_B) \land B : B_i(\neg acceptable(\rho'), t'_B) \land$$
$$\neg \exists p : (p \in \Gamma_i \land p \notin \Gamma'_i) \land I : I_i(succeed(Negotiate(r, Do(r, \rho))), t'_I) \Rightarrow C_{out} : info\_null(i, r, Do(r, \rho), t_{cout,info\_null})$$

where

$$During(t_{cin,counter}, t'_I) \land Meets(t_{cin,counter}, t_B) \land During(t'_B, t'_I)$$
$$\land Finishes(t'_B, t_B) \land Meets(t_B, t_{cout,info\_null}) \land During(t_{cout,info\_null}, t'_I).$$

This axiom is the counterpart of *Info 3*.

Success 2. When an initiating agent (i) receives a counter-offer (ρ') from a responding agent (r), i believes that r desires to perform ρ'. If ρ' is acceptable, then i agrees.

$$C_{in} : counter(r, i, Do(r, \rho), t_{cin,counter}) \land B : B_i(D_r(Do(r, \rho')), t_B)$$
$$\land B : B_i(acceptable(\rho'), t'_B) \land I : I_i(succeed(Negotiate(r, Do(r, \rho))), t'_I)$$
$$\Rightarrow C_{out} : agree(i, r, Do(r, \rho), t_{cout,agree}) \land B : B_i(\neg CanDo(r, \rho), t''_B) \land D : D_i(\neg Do(r, \rho), t_D) \land$$
$$I : I_i(\neg Negotiate(r, Do(r, \rho)), t''_I) \land I : I_i(\neg succeed(Negotiate(r, Do(r, \rho))), t''_I) \land success$$

where

$$During(t_{cin,counter}, t'_I) \land During(t'_B, t'_I) \land Finishes(t'_B, t_B) \land Meets(t_B, t_{cout,agree})$$
$$\land Meets(t'_I, t''_I) \land Meets(t_B, t''_B) \land Starts(t_D, t''_I) \land During(t_{cout,agree}, t''_I).$$

The communicative act *agree* is the encapsulation of composing and sending an AGREE message, in which ⟨*contents*⟩ holds the counter-offer ρ'. With this axiom, as soon as the initiating agent agrees to a counter-offer by the responding agent, it (1) believes that the responding agent cannot do the originally requested task, (2) desires no longer that the responding agent performs the task, (3) intends to negotiate no further regarding the task, and (4) intends no longer to have a successful negotiation regarding the task. However, the negotiation still ends with a *success* tag because even though the initiating agent does not get what it wanted originally, it does obtain a portion of its original request; hence it is a partial success.

Failure 2. When an initiating agent (i) receives a STOP message from a responding agent (r), it believes that the responding agent r does not desire to perform the requested task ρ and thus stops negotiating with r to perform the task, and the negotiation fails.

$$C_{in} : stop(r, i, Do(r, \rho), t_{cin,stop}) \land I : I_i(succeed(Negotiate(r, Do(r, \rho))), t'_I) \Rightarrow$$
$$B : B_i(\neg CanDo(r, \rho), t_B) \land D : D_i(\neg Do(r, \rho), t_D) \land I : I_i(\neg Negotiate(r, Do(r, \rho)), t''_I) \land$$
$$I : I_i(\neg succeed(Negotiate(r, Do(r, \rho))), t''_I) \land rejected$$

where $During(t_{cin,stop}, t'_I) \land Meets(t'_I, t_B) \land Meets(t'_I, t_D) \land Meets(t'_I, t''_I)$. The communicative act *stop* is the encapsulation of receiving and parsing a STOP message. This rule is similar to *Failure 1*. In

addition, it also states that the agent intends to not negotiate. In our current design we do not differentiate between an outright failure (failure type 1) and an opt-out failure (failure type 2)—both end with a *rejected* tag.

Failure 3I. This rule is similar to *Failure 2* except for that it deals with an ABORT message and ends with an *abort* tag.

Failure 4I. This rule is similar to *Failure 2* except for that it deals with an OUT_OF_TIME message and ends with an *out_of_time* tag.

Abort I. When an initiating agent (*i*) no longer intends to negotiate with a responding agent (*r*) to perform a requested task ρ, it aborts the negotiation.

$$I : I_i(\neg Negotiate(r, Do(r, \rho)), t'_I) \Rightarrow C_{out} : abort(i, r, Do(r, \rho), t_{cout,abort}) \wedge$$
$$D : D_i(\neg Do(r, \rho), t_D) \wedge I : I_i(\neg succeed(Negotiate(r, Do(r, \rho))), t''_I) \wedge abort$$

where $During(t_{cout,abort}, t'_I) \wedge During(t_{cout,abort}, t_D) \wedge During(t_{cout,abort}, t''_I) \wedge Finishes(t''_I, t'_I)$. The communicative act *abort* is the encapsulation of composing and sending an ABORT message. This rule says that if an initiating agent aborts a negotiation, then it informs the responding agent.

Out_of_time I. When an initiating agent (*i*) runs out of its allocated time for the negotiation with a responding agent (*r*) to perform a requested task ρ, it aborts the negotiation.

$$B : B_i(\neg time(Negotiate(r, Do(r, \rho))), t_B) \wedge I : I_i(succeed(Negotiate(r, Do(r, \rho))), t'_I) \Rightarrow$$
$$C_{out} : out\_of\_time(i, r, Do(r, \rho), t_{cout,out\_of\_time}) \wedge I : I_i(\neg Negotiate(r, Do(r, \rho)), t''_I) \wedge$$
$$I : I_i(\neg succeed(Negotiate(r, Do(r, \rho))), t''_I) \wedge out\_of\_time$$

where $During(t_B, t'_I) \wedge During(t_{cout,out\_of\_time}, t''_I) \wedge Meets(t'_I, t''_I)$. The *time* predicate encapsulates the acts of obtaining and comparing the time elapsed for the negotiation against the time allocated for the negotiation. The communicative act *out_of_time* predicate is the encapsulation of composing and sending an OUT_OF_TIME message to the responding agent. This rule states that when the agent has run out of time allocated for the negotiation, it no longer intends to negotiate. This is real-time motivated.

No_response I. When an initiating agent (*i*) detects receives no response from a responding agent (*r*) during a negotiation, then it unilaterally quits the negotiation with a failure.

$$B : B_i(no\_response(r), t_B) \wedge I : I_i(succeed(Negotiate(r, Do(r, \rho))), t'_I) \Rightarrow$$
$$I : I_i(\neg Negotiate(r, Do(r, \rho)), t''_I) \wedge I : I_i(\neg succeed(Negotiate(r, Do(r, \rho))), t''_I) \wedge channel\_jammed$$

where $During(t_B, t'_I) \wedge Meets(t'_I, t''_I)$. The *no_response* predicate is one of our real-time enabling functional predicates to be discussed next. This axiom allows an agent to bail out of a negotiation when the negotiation partner fails to respond.

Figure 2 shows the time lines of the initiating agent's behavior when faced with an outright rejection or agreement. These are the simple cases of the axioms above. The length of the process is based on $t_{cout,initiate}$, t_{cin,no_go}, and $t_{cin,agree}$. During such time, $I : I_i\left(Negotiate\left(r, Do(r, \rho)\right), t_I\right)$ holds true. After that, the intention can be removed or modified.
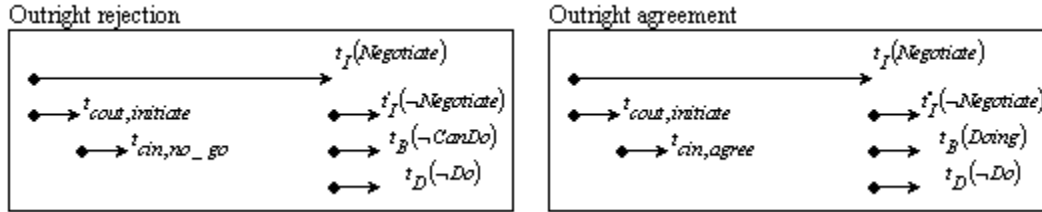

Figure 2 (a) Outright rejection and (b) outright agreement, from the initiating agent's point of view.

Figure 3 shows the negotiation process, from the initiating agent's point of view once the responding agent agrees to negotiate. The negotiation process is a manifestation of the axioms discussed above, proving a flow of communicative acts and BDI states that drives the completion of the negotiation. It is with the temporal BDI axioms that we are able to produce Figure 4, an explicit outline of the interactions of the communicative acts with various BDI states— specifying when and how long certain states must hold true, cannot be modified, can change, can be accessed, or are of no concern. It is also through the axioms that we are able to guarantee the completion of a negotiation process within a certain time. For example, if the initiating agent goes through the following steps: *initiate*, parse *respond*, *info*, parse *more_info*, *info*, parse *more_info*, *info_null*, parse *counter*, check to see whether the counter-offer is acceptable, and agree, then we know how much time it takes to do so by summing up the temporal intervals associated with each step. The acceptability of the counter-offer has to be held constant throughout the agreement step. This explicit declaration of time constraints is important since each of our agents is multi-threaded, where several threads may access and need to modify the same variable at the same time. Without the axioms, a variable such as the acceptability of the counter-offer might *accidentally* be modified by another negotiation thread, rendering the current negotiation process ambiguous. Moreover, with the temporal intervals, we are able to fine-tune the system by observing the time usage of each communicative act for speedup. For example, with the BDI states, we know the minimal time we need to hold the value of a variable constant. The shorter the time needed for a variable to be held constant, the more frequent the variable can be updated and accessed, allowing other threads to proceed.

18
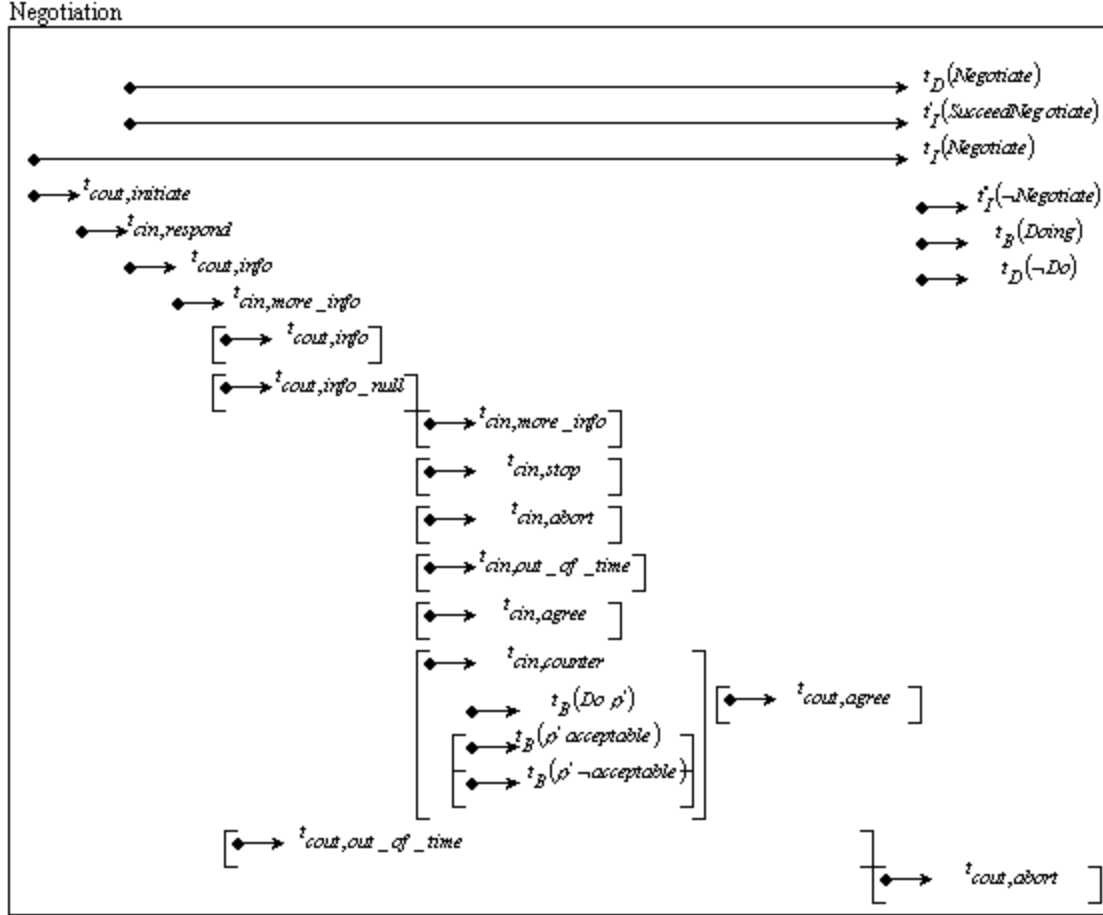
Figure 3 Negotiation from the initiating agent's point of view (a simplified version). Temporal intervals in brackets are options. Note that once the initiating agent receives a RESPOND message from the responding agent, it is committed to negotiate successfully since it believes that the responding agent desires to negotiate.

2.4.2. Responding Behavior

Here we outline axioms that link an agent's communication and its internal states for conducting negotiations as a responding agent.

No_go. When a responding agent (r) believes that it cannot perform a requested task ρ from an initiating agent i, it outright refuses to negotiate.

$$C_{in} : initiate\big(i, r, Do(r, \rho), t_{cin,initiate}\big) \wedge B : B_r\big(\neg CanDo(r, \rho), t_B\big) \Rightarrow$$
$$C_{out} : no\_go\big(r, i, Do(r, \rho), t_{cout,no\_go}\big) \wedge exit$$

where $Meets\big(t_{cin,initiate}, t_B\big) \wedge During\big(t_{cout,no\_go}, t_B\big)$. The communicative act *no_go* encapsulates the act of composing a NO_GO message and sending the message to agent i. Agents are responsible in that if a responding agent refuses to negotiate, it informs the initiating agent.

Agree 1. When a responding agent (r) (1) believes that it is already performing a requested task ρ or (2) desires to perform ρ, it agrees to perform the task.

$$C_{in} : initiate\big(i,r,Do(r,\rho),t_{cin,initiate}\big) \wedge \big(B : B_r\big(Doing(r,\rho),t_B\big) \vee D : D_r\big(Do(r,\rho),t_D\big)\big)$$
$$\Rightarrow C_{out} : agree\big(r,i,Do(r,\rho),t_{cout,agree}\big) \wedge D : D_r\big(Do(r,\rho),t'_D\big) \wedge success$$

where

$$\big(Overlaps\big(t_{cin,initiate},t_B\big) \wedge Before\big(t_{cin,initiate},t_{cout,agree}\big) \wedge During\big(t_{cout,agree},t_B\big) \wedge Meets\big(t_{cout,agree},t'_D\big)\big)$$
$$\vee \big(Overlaps\big(t_{cin,initiate},t_D\big) \wedge Before\big(t_{cin,initiate},t_{cout,agree}\big) \wedge Finishes\big(t_{cout,agree},t_D\big) \wedge Meets\big(t_D,t'_D\big)\big)$$.

The communicative act *agree* encapsulates the acts of composing an AGREE message and sending the message to the initiating agent. The rule *strengthens* the desire of the agent to continue performing the task. As previously discussed, an agent may be still performing a task while it no longer desires to do so because a task may be atomic and non-interruptible. So, when the responding agent realizes the initiating agent requests for the same task to be performed, then it re-asserts its desire to ensure the continuation of the task. We also use the relation $Meets(t_D,t'_D)$ to transition the desire—extending the period of time for the responding agent's desire to perform the task.

Note also that with this rule, the responding agent agrees to help only because it is already performing the task, but the *agree* predicate does not reveal that to the initiating agent. This simplifies our agent design in two ways: (1) the responding agent does not have to explain to the initiating agent why it agrees to perform a requested task, and (2) the initiating agent does not have to remember why the responding agent agreed to a requested task.

Respond. When a responding agent (r) believes that it can perform a requested task ρ, and there is no desire to perform ρ nor belief that it is performing ρ, it responds to the negotiation request, i.e., it agrees to negotiate.

$$C_{in} : initiate\big(i,r,Do(r,\rho),t_{cin,initiate}\big) \wedge B : B_r\big(CanDo(r,\rho),t_B\big) \wedge \neg \exists D : D_r\big(Do(r,\rho),t_D\big) \wedge$$
$$\neg \exists B : B_r\big(Doing(r,\rho),t'_B\big) \Rightarrow C_{out} : respond\big(r,i,Do(r,\rho),t_{cout,respond}\big) \wedge$$
$$I : I_r\big(Negotiate(i,Do(r,\rho)),t_I\big) \wedge I : I_r\big(succeed(Negotiate(i,Do(r,\rho))),t'_I\big)$$

where

$$Meets\big(t_{cin,initiate},t_B\big) \wedge During\big(t_B,t_D\big) \wedge During\big(t_B,t'_B\big) \wedge During\big(t_{cout,respond},t_B\big) \wedge$$
$$Meets\big(t_{cout,respond},t_I\big) \wedge Meets\big(t_{cout,respond},t'_I\big) \wedge Finishes(t'_I,t_I) \wedge Before\big(t_{cin,initiate},t_{cout,respond}\big)$$.

The communicative act *respond* encapsulates the acts of composing a RESPOND message and sending the message to the initiating agent. If the responding agent believes it can perform the task, and yet it currently does not have a desire to do so, and does not believe it is performing the task, then it decides to negotiate. Note that the implicit social behavior of the agents is at play here: because of the cooperativeness of the agent, it intends to negotiate and intends to negotiate successfully. These two intentions motivate the responding agent to continue negotiating.

More_info. When a responding agent (r) receives arguments from an initiating agent (i) for a requested task ρ, it processes the arguments to update evidence support for the task. If

the support is still lacking, and the negotiation is on time or the task is discrete, then it asks for more arguments.

$$C_{in} : info(i, r, Do(r, \rho), t_{cin,info}) \land I : I_r(succeed(Negotiate(i, Do(r, \rho))), t'_I) \land I : I_r(update(\Gamma'_i), t''_I) \land$$
$$\neg \exists D : D_r(Do(r, \rho), t_D) \land (B : B_r(\neg slow(Negotiate(i, Do(r, \rho))), t_B) \lor B : B_r(discrete(\rho), t'_B))$$
$$\Rightarrow C_{out} : more\_info(r, i, Do(r, \rho), t_{cout,more\_info})$$

where

$$Meets(t_{cin,info}, t''_I) \land During(t_{cin,info}, t'_I) \land During(t''_I, t'_I) \land Meets(t''_I, t_D) \land During(t_D, t'_I)$$
$$\land During(t_D, t_B) \land During(t_D, t'_B) \land During(t_{cout\_more\_info}, t_D) \land Before(t_{cin,info}, t_{cout,more\_info}).$$

Auxiliary to More_info. The action *update* examines the arguments collected, Γ'_i during $t_{evidence}$ where $Finishes(t_{evidence}, t''_I)$, to see if the proposition $\Gamma'_i \cup \Gamma_r \models D : D_r(Do(r, \rho), t_D)$. If so, then $D : D_r(Do(r, \rho), t_D)$ where $Meets(t''_I, t_D)$.

First, the communicative act *info* encapsulates the actions of receiving and parsing an INFO-type message from the initiating agent, in which the $\langle contents \rangle$ holds the arguments $p \in \Gamma_i$ during t_{comm} and $p \notin \Gamma'_i$ during t_{comm} where $Before(t_{comm}, t_{cin,info})$. The responding agent then examines the arguments by invoking the predicate *update*. Since *update* is an action, it is self-terminating and $Equal(t''_I, [update(\Gamma'_i)])$. If the arguments are sufficient, then *update* results in $D : D_r(Do(r, \rho), t_D)$; otherwise, the negotiation continues. If at the meantime the agent believes that the pace of the negotiation is not slow or that the task is discrete, then it continues to ask for more information from the initiating agent. Note that in our protocol, a responding agent can only make a counter-offer when the requested task is non-discrete. There are two conditions that prompt a responding agent to counter-offer: (1) when the initiating agent does not have any more arguments (as discussed later), or (2) when the pace of the negotiation is slow. The predicate *slow* measures the pace of a negotiation. The predicates *update*, *slow*, and *discrete* are part of our real-time enabling functional predicates and will be discussed next.

Agree 2. When a responding agent (*r*) receives arguments from an initiating agent (*i*) for a requested task ρ, it processes the arguments to update evidence support for the task. If the support is enough, then it agrees to perform ρ.

$$C_{in} : info(i, r, Do(r, \rho), t_{cin,info}) \land I : I_r(succeed(Negotiate(i, Do(r, \rho))), t'_I) \land$$
$$I : I_r(update(\Gamma'_i), t''_I) \land D : D_r(Do(r, \rho), t_D) \Rightarrow C_{out} : agree(r, i, Do(r, \rho), t_{cout,agree}) \land$$
$$I : I_i(\neg Negotiate(i, Do(r, \rho)), t'''_I) \land I : I_i(\neg succeed(Negotiate(i, Do(r, \rho))), t'''_I) \land success$$

where

$$Meets(t_{cin,info}, t''_I) \land During(t_{cin,info}, t'_I) \land During(t''_I, t'_I) \land Meets(t''_I, t_D) \land$$
$$During(t_D, t'_I) \land During(t_{cout,agree}, t_D) \land During(t_{cout,agree}, t'''_I) \land Meets(t'_I, t'''_I).$$

This axiom is a counterpart of *More_info*. If it turns out that the arguments are sufficient, then the responding agent agrees to the request. It uses the communicative act *agree* to compose and send an AGREE-type message to the initiating agent. The negotiation ends with a *success* tag. Also, the agent also stops intending to negotiate and to negotiate successfully.

Counter 1. When a responding agent (r) receives arguments from an initiating agent (i) for a requested task ρ, it processes the arguments to update evidence support for the task. If the support is not enough, the pace of the negotiation is slow, and ρ involves non-discrete resource, then r makes a counter-offer (ρ').

$$C_{in} : info(i,r,Do(r,\rho),t_{cin,info}) \wedge I : I_r(succeed(Negotiate(i,Do(r,\rho))),t_I') \wedge$$
$$I : I_r(update(\Gamma_i'),t_I'') \wedge \neg \exists D : D_r(Do(r,\rho),t_D) \wedge B : B_r(slow(Negotiate(i,Do(r,\rho))),t_B) \wedge$$
$$B : B_r(\neg discrete(\rho),t_B') \Rightarrow C_{out} : counter(r,i,Do(r,\rho),t_{cout,counter})$$

where

$$Meets(t_{cin,info},t_I'') \wedge During(t_{cin,info},t_I') \wedge During(t_I'',t_I') \wedge Meets(t_I'',t_D) \wedge$$
$$During(t_D,t_I') \wedge Finishes(t_D,t_B) \wedge Finishes(t_D,t_B') \wedge During(t_{cout,counter},t_D).$$

The communicative act *counter* encapsulates the acts of finding a counter-offer (ρ'), composing a COUNTER-type message, and sending the message to the initiating agent. The counter-offer ρ' is stored in the $\langle contents \rangle$ of the message. This is a companion rule to *More_info* as discussed above. If the negotiation is off pace, then instead of asking for more information/arguments, the responding agent counter-offers. This is motivated by the intention of the agent to achieve a successful outcome to the negotiation. However, if the task involves a discrete resource, then a counter-offer is impossible and the *More_info* rule overwrites the *Counter 1* rule. Further, even though the responding agent makes a counter-offer, it does not have the desire to perform the task counter-offered. It only has the desire to do so after the initiating agent agrees to it. This is represented later in axiom *Success 3*.

Note that the motivation behind a counter-offer is to speed up the pace of the negotiation or as a last-ditch effort to salvage a failing negotiation. We do not perform counter-offer as part of the normal interaction—to evaluate and re-plan proposals at each negotiation step would have slowed down our negotiations and that is not applicable to a real-time problem.

Stop. When a responding agent (r) is notified by an initiating agent (i) that it has no more arguments for a requested task ρ, and the task is discrete, the agent r stops the negotiation, and the negotiation fails.

$$C_{in} : info\_null(i,r,Do(r,\rho),t_{cin,info\_null}) \wedge I : I_r(succeed(Negotiate(i,Do(r,\rho))),t_I') \wedge$$
$$I : I_r(update(\Gamma_i'(Do(r,\rho))),t_I'') \wedge \neg \exists D : D_r(Do(r,\rho),t_D) \wedge B : B_r(discrete(\rho),t_B')$$
$$\Rightarrow C_{out} : stop(r,i,Do(r,\rho),t_{cout,stop}) \wedge I : I_r(\neg Negotiate(i,Do(r,\rho)),t_I''') \wedge$$
$$I : I_r(\neg succeed(Negotiate(i,Do(r,\rho))),t_I''') \wedge stop$$

where

$$Meets\left(t_{cin,info\_null},t_I''\right)\wedge During\left(t_{cin,info\_null},t_I'\right)\wedge During\left(t_I'',t_I'\right)\wedge Meets\left(t_I'',t_D\right)\wedge During\left(t_D,t_I'\right)$$
$$\wedge Finishes\left(t_D,t_B\right)\wedge Finishes\left(t_D,t_B'\right)\wedge During\left(t_{cout,stop},t_D\right)\wedge Meets\left(t_I',t_I'''\right)\wedge Starts\left(t_{cout,stop},t_I'''\right)\;.$$

The communicative act *info_null* is the encapsulation of receiving and parsing an INFO_NULL-type message from the initiating agent, while the communicative act *stop* encapsulates the actions of composing a STOP-type message and sending the message to the initiating agent. This is when the responding agent gives up on the negotiation, after being informed that no more arguments are on the way. As a result, it no longer intends to negotiate, and it opts out by informing the initiating agent as a responsible gesture. The proposition *stop* indicates the failure of a negotiation because the responding agent is not convinced to perform the requested task.

Counter 2. When a responding agent (*r*) is notified by an initiating agent (*i*) that it has no more arguments for a requested task ρ, and the task is discrete, the agent *r* makes a counter-offer (ρ').

$$C_{in}:info\_null\left(i,r,Do\left(r,\rho\right),t_{cin,info\_null}\right)\wedge I:I_r\left(succeed\left(Negotiate\left(i,Do\left(r,\rho\right)\right)\right),t_I'\right)\wedge$$
$$I:I_r\left(update\left(\Gamma_i'\right),t_I''\right)\wedge\neg\exists D:D_r\left(Do\left(r,\rho\right),t_D\right)\wedge B:B_r\left(\neg discrete\left(\rho\right),t_B'\right)$$
$$\Rightarrow C_{out}:counter\left(r,i,Do\left(r,\rho\right),t_{cout,counter}\right)$$

where

$$Meets\left(t_{cin,info},t_I''\right)\wedge During\left(t_{cin,info},t_I'\right)\wedge During\left(t_I'',t_I'\right)\wedge Meets\left(t_I'',t_D\right)\wedge$$
$$During\left(t_D,t_I'\right)\wedge Finishes\left(t_D,t_B\right)\wedge Finishes\left(t_D,t_B'\right)\wedge During\left(t_{cout,counter},t_D\right).$$

This axiom is a counterpart of *Stop*, and closely resembles *Counter 1*. Driven by the intention to succeed in the negotiation, the responding agent, after being notified of no more arguments coming in from the initiating agent, voluntarily makes a counter-offer if the task is non-discrete.

Failure 3R. This axiom is similar to *Failure 3I*.

Failure 4R. This axiom is similar to *Failure 4I*.

Abort R. This axiom is similar to *Abort I*.

Out_of_time R. This axiom is similar to *Out_of_time I*.

Success 3. When a responding agent (*r*) receives an AGREE message from an initiating agent (*i*) to its counter-offer (ρ'), the responding agent desires to perform the counter-offer, and the negotiation ends with success.

$$C_{in}:agree\left(i,r,Do\left(r,\rho\right),t_{cin,agree}\right)\wedge I:I_r\left(succeed\left(Negotiate\left(i,Do\left(r,\rho\right)\right)\right),t_I'\right)$$
$$\Rightarrow D:D_r\left(Do\left(r,\rho'\right),t_D\right)\wedge I:I_i\left(\neg Negotiate\left(i,Do\left(r,\rho\right)\right),t_I''\right)\wedge$$
$$I:I_i\left(\neg succeed\left(Negotiate\left(i,Do\left(r,\rho\right)\right)\right),t_I''\right)\wedge success$$

where $Finishes(t_{cin,agree}, t'_I) \wedge Meets(t'_I, t_D) \wedge Meets(t'_I, t''_I)$.   The communicative act *agree* predicate encapsulates the acts of receiving and parsing an AGREE-type message, in which $\langle contents \rangle$ holds the information regarding ρ'. This rule says that if the initiating agent agrees to the counter-offer, then the responding agent (1) desires to perform the counter-offered task and (2) intends no longer to continue with the negotiation regarding the originally requested task.

No_response R. This is similar to *No_response I*.

Figure 4 shows the time lines of the initiating agent's behavior when faced with an outright rejection or agreement. These are the simple cases of the axioms above. See for example that when the responding agent agrees to a request, it extends the desire to do the requested task.
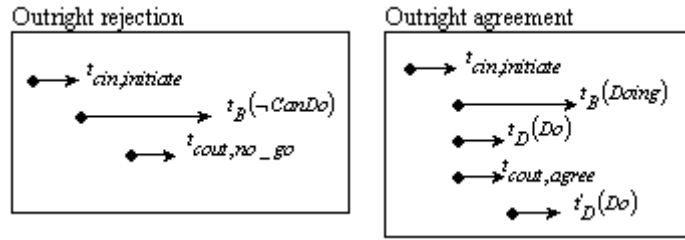

Figure 4 (a) Outright rejection and (b) outright agreement, from the responding agent's point of view.

Figure 5 shows the negotiation process, from the responding agent's point of view once it agrees to negotiate. Similar to Figure 3, Figure 5 allows us to explicitly describe the temporal relationships among the BDI states and the communicative acts. That description, in turn, allows us to guarantee the behavior of the negotiation and to fine-tune its efficiency.

Functional Predicates
In this section, we describe the functional predicates mentioned in the previous section that present the logical framework for the rules of encounter between two agents. These predicates are the infrastructure to our real-time negotiation protocol. To simplify the discussion, we have touched upon 11 communicative acts such as *initiate*, *respond*, *no_go*, *agree*, etc. and six negotiation-related functional predicates: *slow*, *time, discrete, no_response, acceptable*, and *update*. Here we will elaborate further on the six predicates as they are an integral part that enables the real-time negotiation between the agents.
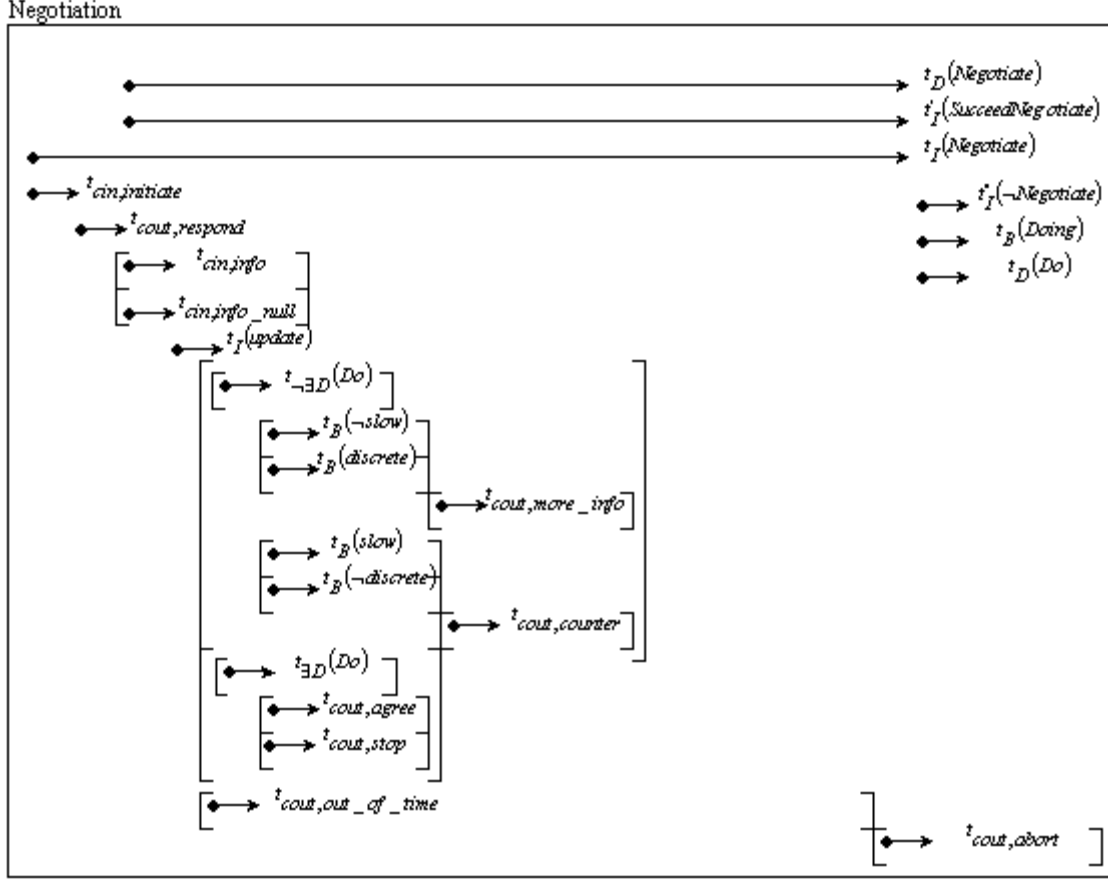
Negotiation

$t_D(Negotiate)$

$t_I(SucceedNegotiate)$

$t_I(Negotiate)$

$t_{cin,initiate}$

$t_{cout,respond}$

$t_{cin,info}$

$t_{cin,info\_null}$

$t_I(update)$

$t_{\neg\exists D}(Do)$

$t_B(\neg slow)$

$t_B(discrete)$

$t_{cout,more\_info}$

$t_B(slow)$

$t_B(\neg discrete)$

$t_{cout,counter}$

$t_{\exists D}(Do)$

$t_{cout,agree}$

$t_{cout,stop}$

t_{cout,out_of_time}

$t_I(\neg Negotiate)$

$t_B(Doing)$

$t_D(Do)$

$t_{cout,abort}$

Figure 5 Negotiation from the responding agent's point of view (a simplified version). Temporal intervals under brackets are options. Note that once the responding agent sends out a RESPOND message to the initiating agent, it is committed to negotiate successfully.

slow

This predicate takes the form of *slow*(*action*), and given an action (or a task), it measures the pace of the action and returns true or false. An action has two temporal intervals: the actual real-time interval, [*action*], and the planned/predicted interval, ‖*action*‖. Suppose that [*action*] has a duration between $t_{s,action}$ and $t_{f,action}$, and the set of states as a result of the action is $S_{action} = \{s_{0,action}, \cdots, s_{N,action}\}$. Each state, $s_{n,action}$, holds during an actual interval $\lfloor s_{n,action} \rfloor$ such that (1) *During*($\lfloor s_{n,action} \rfloor$, [*action*]), (2) *Overlaps*($\lfloor s_{n,action} \rfloor$, $\lfloor s_{n+1,action} \rfloor$), and (3) the temporal interval of two such states has a duration between the start of $s_{n,action}$ and the latest finish time of the two states, $\lfloor s_{n,action} \; s_{n+1,action} \rfloor$. Similarly, we can obtain $\| s_{n,action} \; s_{n+1,action} \|$

When *slow*(*action*) is invoked, it retrieves the current state of the action, $s_{current,action} \in S_{action}$. If

$$\frac{[s_{0,action} \; s_{current,action}]}{[action]} > \frac{\| s_{0,action} \; s_{current,action} \|}{\| action \|}$$

is true, then *slow*(*action*) returns true; otherwise, it returns false. Note that this predicate is binary since we use it to trigger a *counter_offer* act. A more general approach is to use a *degree* of

25

slowness that would not only trigger a *counter_offer* but also dictate how conceding the responding agent should be.

time
This predicate takes the form of *time*(*action*) and indicates whether an agent has run out of the allocated time for the *action*. Suppose we have [*action*] and ‖*action*‖. When an agent invokes *time*(*action*), the predicate measures the time elapsed so far, $t_{elapsed} = t_{current} - t_{s,action}$. If $t_{elapsed} \geq \|action\|$, then *time*(*action*) returns true; otherwise it returns false.

discrete
A discrete task is when the task cannot be broken up or attenuated. For example, if the initiating agent asks the responding agent to turn on a sensor, then the responding agent may only respond with yes or no. However, if the initiating agent asks the responding agent to turn on a sensor in five seconds, then the responding agent, in addition to yes or no, may also counter with "yes, but in 10 seconds". The introduction of the time factor makes the task a non-discrete one, allowing the responding agent to counter-offer. In our negotiation protocol, a task ρ may be qualified by time and resource amount. The predicate *discrete* returns true if both qualities are absent, otherwise, it returns false. Note that we intentionally leave out the qualification of tasks to simplify our discussion.

no_response
This predicate takes one argument—the agent from which the current agent is waiting for a message. In general, when an agent *i* invokes $no\_response(r)$ on another agent *r*, it is after agent *i* has performed a communicative act, *C*. Hence, $Before([C],[no\_response(r)])$. If agent *i* does not receive a response in the interval t_{window} (and $During(t_{window}, t_I)$ given that the intention *I* is to negotiate), then $no\_response(r)$ returns true; otherwise, it returns false.

acceptable
This predicate is only invoked by an initiating agent, *I*, and takes as argument a task. Suppose the agent *i* desires to achieve goal *G*. To achieve *G*, there is a set of subtasks $\vec{\rho} = \{\rho_1, \rho_2, \cdots, \rho_{|C_i(\vec{\rho})|}\}$, and a coalition, $C_i(\vec{\rho})$, exists to distribute the subtasks among the coalition members. As a result,

$$D_i(G, t_D) \Rightarrow D_i(Do(C_i(\vec{\rho}), \vec{\rho}), t_D),$$

where $D_i(Do(C_i(\vec{\rho}), \vec{\rho}), t_D) = \left\{ D_i(Do(r_1, \rho_1), t_{D1}), \cdots, D_i\left(Do\left(r_{|C_i(\vec{\rho})|}, \rho_{|C_i(\vec{\rho})|}\right), t_{D|C_i(\vec{\rho})|}\right) \right\}$. As the agent progresses in real time, *G* may become *G′* such that $D_i(G', t'_D) \Rightarrow D_i(Do(C_i(\vec{\rho}'), \vec{\rho}'), t'_D)$ where $During(t'_D, t_D)$. For example, an agent has received commitments for some negotiated resources, so it no longer desires the original set of resources it asked for. So, when the initiating agent receives a counter offer ρ' from the responding agent, it invokes *acceptable*(ρ') to compare the counter-offered task ρ' with the corresponding ρ_r where $D_i(Do(r, \rho'_r), t'_D)$. If $\rho' \in \rho_r$ then *acceptable*(ρ') returns true; otherwise, it returns false.

update
This functional predicate is invoked only by a responding agent *r*, as previously mentioned in an auxiliary to the *More_info* axiom.

2.5. Implementation

The driving application for our system is multisensor target tracking, a distributed resource allocation and constraint satisfaction problem. The objective is to track as many targets as possible and as accurately as possible using a network of sensors. Each sensor has a set of consumable resources, such as beam-seconds (the amount of time a sensor is active), battery power, and communication channels, that each sensor desires to utilize efficiently. Each sensor is at a fixed physical location and, as a target passes through its coverage area, it has to cooperate with neighboring sensors to triangulate their measurements to obtain an accurate estimate of the position and velocity of the target. As more targets appear in the environment, the sensors need to decide which ones to track, when to track them, and when not to track them, always being aware of the status and usage of sensor resources. Each sensor can at any time scan one of three sectors, each covering a 120-degree swath. Sensors are connected to a network of CPU platforms on which the agents controlling each sensor reside. The physical sensors are 9.35 GHz Doppler MTI radars that communicate using a 900 MHz wireless, radio-frequency (RF) transmitter. The agents (and sensors) must communicate over an eight-channel RF link, leading to potential channel jamming and lost messages. Our agents may reside on the same CPU platform or different platforms.

We have implemented our multiagent system in C++. Each agent has 3+*n* threads: (1) a core thread that performs the reasoning, message checking, task handling chores of the agent and thus is always active, (2) a communication thread that is responsible for polling for incoming messages and sending out messages and thus is always active, (3) an execution thread that performs sensor-related tasks such as calibration, target searching, and tracking, and thus is sometimes active and sometimes dormant, and (4) *n* negotiation threads that each waits to be awaken to perform a negotiation and goes back to a dormant state after the negotiation is over. This setup allows an agent to carry out various lines of tasks concurrently. It also allows an agent to conduct multiple negotiations in parallel. Each negotiation thread can be an initiating thread or a responding thread, depending on the dynamic, real-time instructions given by the core thread.

The architecture of our agents is able to detect a target, forms a coalition, performs CBR to determine its negotiation strategy, initiates or responds to negotiations, argues to persuade its partner to perform a task or reasons to whether to agree to perform a task, monitors its own status such as its sensor, noise, tasks, and CPU resource usage, interacts with either a software simulation or the actual physical hardware setup, and obtains real-time data from the operating system supporting its execution. More importantly, each agent is autonomous and can sense and react to real-time events in the environment. Moreover, there is no hierarchy within the multiagent system—all agents are peers.

We have also implemented the complete real-time argumentative negotiation protocol in the negotiator module of an agent. With the formalisms encoded, the negotiator conducts a

negotiation with high efficiency and autonomy. We have implemented all communicative acts: parsing messages and converting them to belief states and converting belief states and composing messages out of them. We have also implemented the belief, desire, and intention states as functions, procedures, and clauses. As for the temporal definitions and constraints of those states, we have implemented recursive mutexes (semaphore-like designs) to manage read/write accesses: when to acquire the value of a state, when to release a lock on the value of a state, when must a state be ready in order for a certain task to start, and so on. For example, an agent may be tracking a target and negotiating at the same time. While tracking, the agent may realize that the target is no longer visible. This directly affects the on-going negotiation since now the agent's sensor has become available, leading to a lower threshold, for example, of a counter-offer. But, the negotiator module of a negotiation thread cannot afford to constantly check the states of the tracking. It does so occasionally and only when it is necessary, and can only be interrupted at certain points over the course of the negotiation, dictated by the temporal definitions of the states.

2.5.1. Real-Time Scheduling Service (RTSS)

We have implemented a Real-Time Scheduling Service (RTSS) in 'C', on top of the KU Real-Time system (KURT) (Srinivasan *et al.,* 1998) that adds real-time functionality to Linux. First, the RTSS provides an interface between the agents and the system timers, allowing agents to: (1) query the operating system about the current time; (2) ask the RTSS to notify them after the passage of certain length of time; and (3) ask the RTSS to ping them at fixed time intervals. This allows agents to know when to, for example, conclude a negotiation process or turn on a radar sector. Second, the agents may ask the RTSS to notify them when certain system-level events occur, such as process threads being activated, or communication messages going out or coming into the system. Third, the agents can ask the RTSS to allocate them a percentage of the CPU for each one of their threads (such as the ones controlling the radar and tracking or the ones used in negotiations) and to schedule this allocation within an interval of time. This RTSS allows an agent to monitor the progress of its own negotiations and the usage status of its allocated CPU resource.

2.5.2. Case-Based Argumentative Negotiation

We have implemented the CBR Manager to maintain the case base of an agent. The implementation includes similarity-based retrieval, both difference- and outcome-driven adaptations, and the incremental and refinement learning. We have implemented the entire negotiation protocol, as depicted in Figure 1, into each negotiation thread. Each thread is capable of monitoring the pace of its own negotiation, retrieving messages via the communication thread, parsing incoming messages, making decisions and reasoning, composing an outgoing message and sending messages via the communication thread. Each thread is autonomous in a way that the core thread of the agent does not have to tell the thread how to conduct a negotiation once it has gotten underway.

Each thread forks off a child process that automatically invokes CLIPS. The communication between a negotiation thread and its child CLIPS process is through pipes. After receiving an acknowledge signal from the CLIPS child process, the negotiation thread informs the core thread that it is ready to accept a negotiation task and waits. When it finally receives an activation signal, the negotiation thread downloads the relevant information regarding the negotiation task. From the information, the thread decides its identity—either an initiating thread or a responding

thread. Then the negotiation thread negotiates following the negotiation protocol described in Section 3. Once the negotiation is done, the thread updates its status and waits for a signal from the core thread before resetting itself for the next negotiation task. Meanwhile, the core thread of the agent periodically checks the status of the active negotiation threads. If a negotiation is completed, the core thread downloads the updated data and signals the negotiation thread that it is okay to reset.

Here we briefly discuss our case-based strategy selection approach. A negotiation strategy dictates a set of tactics for a negotiation. For example, an initiating agent needs to know which arguments are more important to send over first to the responding agent. We use case-based reasoning (CBR) to help us determine that. When an agent encounters a negotiation problem, it searches its casebase for the most similar case, in which the problem description in that case resembles the current negotiation problem. Then, based on the differences between the two problem descriptions, the CBR module of the agent performs an adaptation on the solution. The modified solution becomes the negotiation strategy.

2.5.3. Real-Time Enabling Functional Predicates

In Section 3.4, we presented the logical model of our real-time enabling functional predicates. Here, we describe the implementation that, even though is domain- and application-specific, may serve as a useful example to other designs of the predicates. In this section, we also describe how our agents make a counter-offer in real-time. Note that in our implementation, we employ case-based reasoning to derive a negotiation strategy for an agent for each negotiation task. A case has a set of belief states (situated input parameters), a set of desires (a parametric negotiation strategy), and the outcome of the negotiation. In addition, in our agent design, a negotiation is handled by one of the negotiation threads that an agent dispatches. So, in the following, we will use the term "negotiation thread" quite often and make use of the belief and desire states.

slow

In a case, the desires include the number of negotiation steps allowed and the time allowed. That is, a negotiation thread desires to complete a particular negotiation in n iterations and s seconds. A small n means that fewer messages are exchanged and the negotiation may avoid incurring too much overhead cost per transmission. A small s means that the negotiation is to be completed in a short time. Suppose we denote the number of negotiation steps allowed as $step_{allowed}$, the time allowed as $time_{allowed}$, the number of steps performed so far as $step_{sofar}$, and the time elapsed so far as $time_{sofar}$. We define the *slow* predicate for a responding agent α's negotiation (intending to achieve the desire for performing a requested task ρ) as:

$$slow\big(I_\alpha\big(D_\alpha\big(Do\big(\alpha,\rho\big)\big)\big),t\big)=\left(\frac{time_{sofar}}{step_{sofar}}>\frac{time_{allowed}}{step_{allowed}}\right)$$

time

To implement this predicate, a negotiation thread registers its process ID with a real-time system-level service and makes use of a time-based notification mechanism. A negotiation asks the notification mechanism to signal the thread after s seconds. One unique characteristic of the

mechanism is that the negotiation thread, after registration, may find out how much of the *s* seconds has elapsed after the notification was first registered (for example, 25%, 50, 75%, 100%) by consulting the notification flag: t_{flag}. The value of *s* is determined by the $time_{allowed}$ of the desire states of a case. In our current design, we define the *time* predicate for an agent α's negotiation as:

$$time(negotiation) = (t_{flag} < 1)$$

When $t_{flag} = 1$, that means the time elapsed has reached 100% of $time_{allowed}$.

discrete

First, we denote the set of discrete tasks Θ_{dis} and the set of non-discrete tasks Θ_{con}. Then we define the *discrete* predicate of a requested task ρ as:

$$discrete(\rho) = (\rho_{request} \in \Theta_{dis})$$

where $\rho_{request} \in \rho$ is part of the requested task.

no_response

Our implementation is the following: After an agent sends out a message, it polls its message queue for a response before moving on. After $t_{polling}$ seconds, if the negotiation thread receives no messages from a particular negotiation partner, then *no_response* returns true. If there is a consistently-typed message, then the negotiation thread reacts to it based on our negotiation protocol.

acceptable

This predicate is used only by an initiating agent when it receives a counter-offer from a responding agent and refers to non-discrete (continuous) tasks, Θ_{con}. Let us denote a continuous task as $\bar{r} \in R_{con}$. There are three key parameters in ρ_{con}: $\rho_{con} = \{\rho_{con,name}, \rho_{con,res}, \rho_{con,amount}\}$ where $\rho_{con,name}$ is the name of the task, $\rho_{con,res}$ designates the resource involved in the task, and $\rho_{con,amount}$ indicates the amount of the resource involved.

In our design, when an agent α realizes $B : B_\alpha\left(\rho_{con,amount}^{needed}\right)$ where $\rho_{con,amount}^{needed}$ is nonzero, it initiates negotiations—each with a different $\rho_{con,amount}^{requested}$, to the coalition members—attempting to obtain enough $\rho_{con,amount}$ from the members to meet $\rho_{con,amount}^{needed}$, i.e., $\sum_{C_\alpha(\bar{\rho})} \rho_{con,amount}^{requested} \geq \rho_{con,amount}^{needed}$. At each agent cycle, the agent α updates its $B : B_\alpha\left(\rho_{amount}^{needed}\right)$ and $B : B_\alpha\left(\sum_{C_\alpha(\bar{\rho})} \rho_{con,amount}^{requested}\right)$. If $B : B_\alpha\left(\rho_{con,amount}^{needed} = 0\right)$ then it has achieved its target, and it can abort all current negotiations associated with that particular resource. The process checks (1) the current usage, (2) the anticipated usage, (3) the current allocation, and (4) the agreed additional allocation. As $\rho_{con,amount}^{needed}$ gets smaller, the

remaining negotiations become less demanding in their requests, and vice versa. As each negotiation completes gradually, $\sum_{C_\alpha(\vec\rho)} \rho_{con,amount}^{requested}$ changes.

Suppose that the kth negotiation thread of the agent α is negotiating to obtain $\rho_{con,amount,k}^{requested}$ from a responding agent and the responding agent has just counter-offered ρ_k' with an offered amount of $\rho_{con,amount,k}'$. Then the acceptability of the counter-offer ρ_k' is defined as:

$$acceptable(\rho_k')=\left(\sum_{C_\alpha(\vec\rho)} \rho_{con,amount}^{requested} - \rho_{con,amount,k}^{requested} + \rho_{con,amount,k}' \geq \rho_{con,amount}^{needed} \right)$$

update

The *update* predicate is used when a responding agent receives information or arguments from an initiating agent. The objective of this predicate is to find out whether the evidence support for a requested task is convincing enough for the responding agent to perform it. One key parameter of a negotiation strategy of a responding agent, r, is the persuasion threshold, $T_{persuasion,Do(r,\rho)}$, for a requested task ρ. This is a value created by the responding agent r for ρ; to agree to $Do(r,\rho)$, the arguments sent by the initiating agent must provide for ρ have to be greater than $T_{persuasion,\rho}$.

Suppose we denote the evidence support for $Do(r,\rho)$, with a persuasion threshold, $T_{persuasion,Do(r,\rho)}$, at temporal interval t as $Support(\rho,T_{persuasion,Do(r,\rho)},t)$, and $Support(\rho,T_{persuasion,Do(r,\rho)},time0)=0$. The objective of the initiating agent is to obtain $Support(\rho,T_{persuasion,Do(r,\rho)},t) \geq T_{persuasion,Do(r,\rho)}$ in order to convince the responding agent to perform the requested task. Thus we have the following axiom:

Axiom Desire to Do: If a responding agent r is negotiating with an initiating agent i regarding a requested task ρ, and at temporal interval t, it has $Support(\rho,T_{persuasion,Do(r,\rho)},t) \geq T_{persuasion,Do(r,\rho)}$ then $D:D_r(Do(r,\rho),t)$. (This supplements the *More_info*, *Agree 2*, *Counter 1*, *Stop*, and *Counter 2* axioms)

When arguments are received by the responding agent, the support value changes based on the following agent behavioral model:
  (1) Agents that have cooperated before will tend to cooperate again.
  (2) A responding agent is willing to trust an initiating agent's perception.
  (3) An agent is more inclined to help another agent if that another agent has relied on the agent for help before.
  (4) An agent is more inclined to help another agent if it knows that it is one of the few possible solutions to the requested task.
  (5) An agent is more inclined to help another agent if it knows that that another agent is busy.

Note that the persuasion threshold and the evidence support work together as a joint intention between the two negotiating agents. On one hand, the responding agent is helpful and desires to help the initiating agent, but it also intends to help when it is worthwhile. This implies cooperativeness with a touch of selfishness in a local sense that translates into global optimization of resource allocation. To

make sure that the requested task is worthwhile to do, the responding agent uses a persuasion threshold, derived from its past experience and its current status. On the other hand, the initiating agent intends that the responding agent help with its task. It collects its belief states and sends over whatever it thinks are useful as arguments for its intention. These arguments modify an evidence support value. Therefore, Definition 4 merges the two intentions, seen from two different perspectives, and the joint intention is to achieve a successful negotiation. This deviates from the model proposed by Cohen and Levesque (1990, 1991) but if we view *achieving a successful negotiation* as a *team action*, then both members of the team (the two negotiating agents) are jointly committed to completing the so-called team action, and are mutually believing that they are doing it.

We have implemented the *update* function as a CLIPS-based operation. Evaluation heuristics are coded as CLIPS rules and arguments received from an initiating agent are fed into the CLIPS process (associated with each negotiation thread) to re-compute the evidence support for a requested task. The CLIPS process then sends back the updated evidence support to its negotiation thread and waits for another set of arguments. The negotiation thread and the CLIPS process communicate via a synchronized pipe connection. The negotiation thread compares the updated evidence support with its persuasion threshold. If the former is greater than or equal to the latter, then the negotiation thread agrees to the requested task, composes a message to notify the initiating agent of the deal, and completes its own negotiation. The core thread of the agent then schedules the requested task in its activity.

2.5.4. Counter Offer

When dealing with continuous resources, the responding agent has a persuasion function, modified by two parameters: (1) *kappa* – a willingness factor, and (2) *beta* – a conceding factor, and bounded by the maximum resource that it is willing to give up, $\rho_{con,amount}^{max}$. An agent can use any function[2] to express the continuous persuasion value; in our implementation, we examined two: a linear and an exponential persuasion function.

In our model, the linear persuasion function is:

$$P_{linear}(\tau) = beta \cdot \tau + kappa$$

and, the exponential persuasion function is:

$$P_{exp}(\tau) = kappa \cdot e^{-\tau/beta}$$

The variable τ is the evidence support collected so far, i.e., $\tau = Support(\rho, T_{persuasion,Do(r,\rho)}, t)$. So, in the beginning, at t_s when the support is zero, a responding agent is willing to give up $P_{linear}(0) = kappa$ or $P_{exp}(0) = kappa$. That is why *kappa* is called the willingness factor. Then for $\tau > 0$, the conceding rate depends on *beta*: the larger this value is, the more conceding the

---

[2] There have been a variety of persuasion functions used in previous work in negotiation. Lander and Lesser (1992) used linear functions of local utility values over contract prices. Zlotkin and Rosenschein (1996) suggested nonlinear and exponential worth (or utility) functions and task-based, pre-defined worth functions. In (Faratin et al. 1998), polynomial, exponential, Boulware tactics and Conceder are used.

persuasion function is. The key differences between our persuasion functions and others are (1) the conceding and willingness factors are determined by past experiences, and adapted to fit the current situation, (2) each function applies to an evidence support based on arguments, (3) each function is implicitly bounded by $time_{allowed}$ for a negotiation, and (4) each function is bounded by $\rho^{max}_{con,amount}$, the maximum amount of a resource that the agent is willing to give up.

Thus, when a responding agent is about to make a counter-offer, it checks τ, and the counter-offer, $\rho'_{con,amount}$ is given as:

$$\rho'_{con,amount} = \begin{cases} P(\tau) \\ \rho^{max}_{con,amount} & if\ P(\tau) > \rho^{max}_{con,amount} \end{cases}.$$

2.6. Results

Our experiments concentrated on evaluating whether negotiating agents can track targets better. Our hypothesis was that negotiating agents can track targets better since they can coordinate radar measurements and achieve better triangulation. Our experiments support this hypothesis. In addition to the accuracy of tracking, we used communication as a measure of quality (length of messages, frequency of messages and message cost, i.e. length times frequency), since communication is an important bottleneck to scalability. Here we discuss briefly some experiments, focusing on the benefits of negotiations and case-based reasoning.

First we compared our system to a multiagent, sensor-controlling network where there is no communication between the agents, and where when a target appears in the coverage area of a sensor it is tracked. Next, we compared our case-based negotiating agents to a system where negotiation uses a predefined, static strategy. We selected the static strategy carefully to make sure it should be adequate for most cases. Basically, we used the case that had been frequently retrieved in our previous experiments as the only case in the case-base.

In general, the results, summarized in figures 6-8, were positive. The agents which used no negotiation sent almost 20% more messages, although they had a 50% smaller message cost, but had almost 27% worse tracking accuracy than negotiating agents. The non-negotiating agents exchanged no messages, and only sent their radar measurements to the tracking software. Since there was no coordination of the measurements, there were too many messages sent to the tracker. On the other hand, such messages are short compared to arguments exchanged between agents during negotiation, resulting in lower message costs. Since there was no cooperation to triangulate measurements, the resulting accuracy was poor. The agents that used a static negotiation strategy fared worse than the ones that used a case-based, adaptive strategy. Specifically, the agents using a static protocol sent approximately 10% fewer messages but had a higher message cost, and had almost 18% worse accuracy than the case-based negotiating agents. The message cost is due to the fact that the case-based agents change the ranking of the arguments they communicate based on the situation; this leads to overall more effective communication acts. The accuracy is due to the fact that case-based agents adapt their negotiation to the current situation and have a higher chance of achieving agreement for resource allocation; on the other hand the static strategy agents failed to agree more often and this led to

failure to perform the multiple, simultaneous radar measurements that are required for accurate tracking.
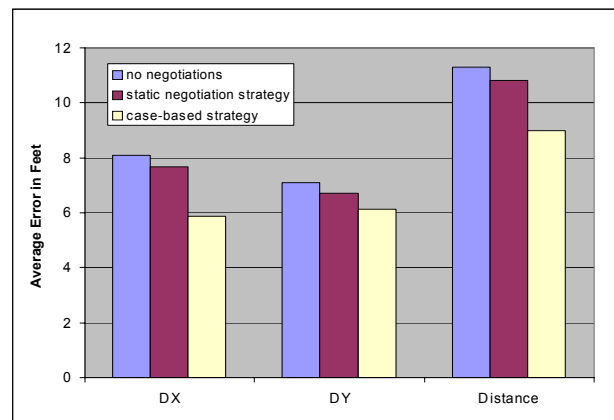

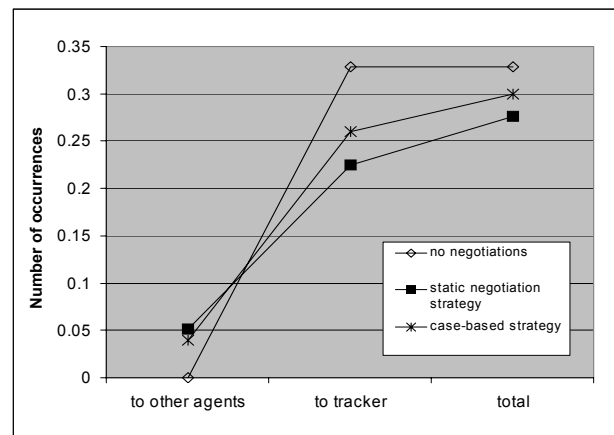**Figure 6** Tracking accuracy vs. agent behavior

Figure 7 Number of messages to agents and to tracking software vs. agent type. Numbers of messages normalized for better comparison.
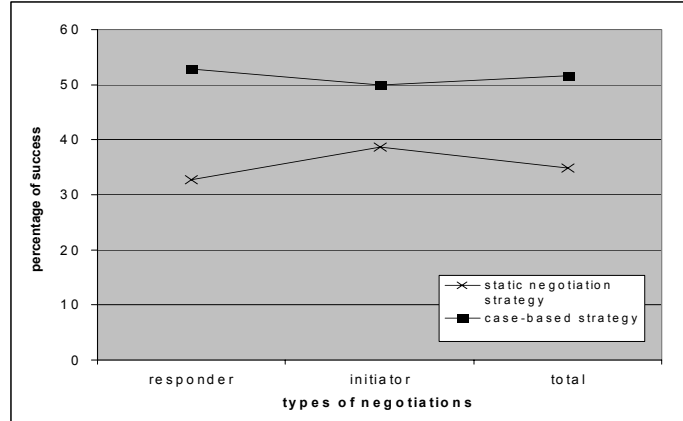
Figure 8 Percentage of successful negotiations vs. negotiation strategy type.

3. Near-Real-Time Negotiating Agents using Domain Constraints

As the project progressed, our choice of using C++ for the agent design became a liability and limitation: using a JNI proxy for communicating with Radsim and the CP code slowed down our code; not being able to run within the CP JVM made it impossible to use RF and only 8 communication channels; finally, there was no graceful way to automatically spawn Tracker agents to respond to the dynamic addition of targets to the environment. Further, as the Challenge Problem (CP) definition shifted more towards tracking than the real-time aspects of resource allocation, it was no longer necessary to integrate with a real-time Linux operating system, and the response of our agents needed to be only near-real-time. Finally, we included simple knowledge about the domain of application which made our implementation less general, but also more realistic.

The underlying methodology of our approach remains the same under the second architecture, and the reader is referred to Section 3 for a description of our theoretical approach. The agents examine their environment and based on it they establish behavioral and negotiation parameters. The agents are divided into two categories: Radar Agents (RA) and Tracker Agents (TA). A Radar Agent controls a sensor, and a Tracker Agent contains the Tracker software. There is a RA for each sensor and a TA for each target. When the RA has no measurement tasks to carry out on behalf of any TA, it searches in all of its sectors to detect targets in round-robin fashion, and it sends these measurements to TAs who would be interested in them. We call this mode the "Search & Detect" mode, and this mode corresponds to the TA's corresponding mode of operation. In this mode, the TA continually checks if the measurements it receives from the RAs it is collaborating with have high-enough confidence for the TA to believe there is a target on a given sector such that the TA should switch to its tracking mode.

The second mode a Radar Agent works in is called the "Measure" mode, and this mode corresponds to the tracking mode of operation of the Tracker Agent. When the TA believes that there is a target visible from a sector on a given RA, it asks for measurements from that RA. So, as soon as the RA has a specific measurement task to perform, it suspends all searching and detecting, and it executes its specific measurement tasks only. Since multiple TAs may ask for

35

measurements from the same sector, the RA keeps track of which TAs have active measure tasks to perform on each of its sectors, and uses negotiation-based techniques to switch between these requests, trying to balance the use of resources.

Tracker Agents provide RAs with information about targets to allow the RAs to reason about which tracking task to schedule, for how long, and how to switch between tracking tasks.

One addition we have made to our methodology is that we have introduced domain-specific reasoning in the Tracking Agents. The TAs use heuristics to evaluate the quality of measurements received from the sensors. Currently, our heuristics integrate the amplitude value of a measurement, its support by multiple sensor sectors, and the expected location of a target compared to where the measurement says the target is.

We also introduced the dynamic instantiation of tracking agents when a potential new target is detected and eventually destroyed when additional measurements cannot be collected to confirm that the target remains active in the scenario-defined "room". Figure 9 shows our program dynamically tracking the two targets from one of the May 2002 CP experiments. The numbers indicate the trackers generated; "b" indicates the beginning of the tracking (when a tracker is generated) and "e" the end (when a tracker loses a target and is destroyed). In figure 9 the agents lost the outside target (indicated by 1b-1e) and reacquired it later (3b-3e).
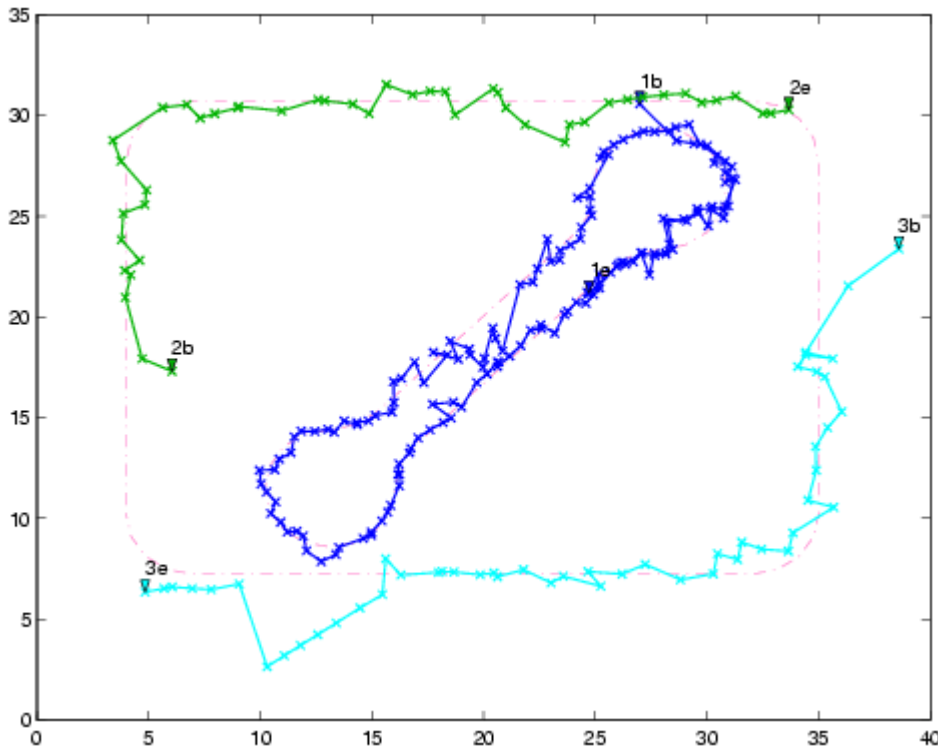

Figure 9: Dynamic allocation of trackers to targets; May 2002 CP set-up.

The majority of the work for these extensions were limited to the definition of a new class, NodeController, that was able to utilize the core classes, RadarAgent and TrackerAgent, in the

initial version of our agent code with only minor changes and extensions. The NodeController class now analyzes Search&Detect measurements from its local RadarAgent and communicates with other NodeControllers to determine if a measurement indicates a new target or supports an already existing target being tracked. If it is a new target, the NodeController instantiates a new TrackerAgent to begin tracking the target and advertises the new tracker to other NodeControllers.

We designed and ran a number of experiments in Radsim, constantly increasing the number of sensors and targets. We have examples of experiments with 3, 5 and 6 targets for which we have been able to generate respectable simulation results using up to 20 active nodes. We have found that it becomes very difficult to design experiments with more than 2 or 3 targets that interact closely *and* satisfy the various ANTS constraints. For example, ensuring that all targets are visible to at least three independent radars and that no more than one target appears in the range of any radar at the same time, is particularly difficult. Our visualization tools helped us analyze and design multi-target configurations. Figure 10 shows one of our two largest experiments to date. The tracks of the targets are indicated by the red figure 8's.
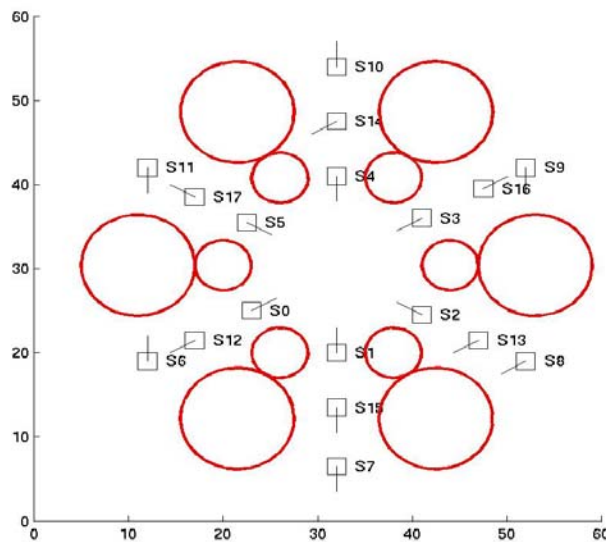

Figure 10: Experiment configurations with 6 targets and 18 sensors.

Figure 11 shows tracking of 6 targets by 18 sensors, using dynamic tracker generation. As the number of sensors and trackers increases, we experience performance degradation in Radsim and the Tracker software that makes the overall system slow down, resulting in worse tracking.
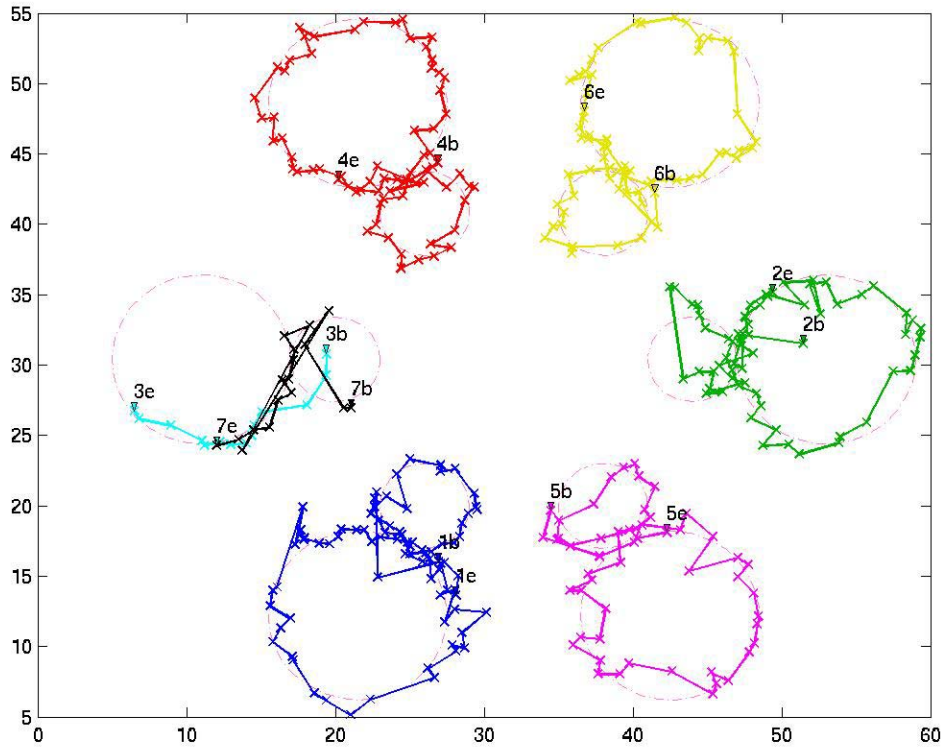
Figure 11: Tracking of six targets. One target was lost and later reacquired (3b-3e, 7b-7e), but most targets are tracked continuously.

4. Challenge Problem Tools and Studies

In addition to the major goal of negotiated resource allocation in distributed domains, our group was also tasked with some related, parallel work.

4.1. The KU Real Time System for Linux (KURT and RTSS)

KURT was expanded and greatly enhanced under this contract. Previously KURT existed only as a kernel patch, with all KURT routines implemented as system calls. Adding or modifying routines required a recompile of the kernel, which could get tedious, especially because each test machine had to be rebooted with the new kernel. Under the contract we ported KURT to a loadable Linux kernel module and eliminated the need for kernel recompiles. Another added benefit is that when changes are necessary, the KURT module can be removed, recompiled, and reinstalled all without requiring a machine reboot. All of this made writing real-time scheduling routines for ANTs far simpler than had we used the strict kernel patching method.

In order to request system resources, we developed a middleware to the agents and the system. This middleware is called RTSS (Real Time Scheduling Services). This layer takes in requests from the agents for system resources. System resources means:

       a. CPU resource

      b. Memory Bandwidth
      c. Network Bandwidth.

Right now the RTSS can guarantee only CPU percentage to the agents. The RTSS, on request from the agent for the CPU, guarantees this percentage of the CPU with the help of KURT. The actual usage of the CPU can be obtained by querying the RTSS about the percentage of the CPU that agent has actually got. This CPU usage information is obtained by modifying the kernel using a macro that gathers information about the registered processes. This helps in reducing the overhead of a function call. In order to improve the accuracy of the KURT real-time scheduling, several optimizations were made to UTIME. UTIME is the microsecond resolution timing patch to Linux that KURT relies upon to schedule real-time processes at the precision of 10's of microseconds. The optimizations included an improved calibration routine that more accurately calculated several CPU speed-specific values at boot time, reducing the execution time of the UTIME timer interrupt handler routine, and a change to how UTIME calculates and executes Linux jiffy events (a jiffy is a heartbeat in Linux) to better accommodate future efforts involving NTP time-standard clock synchronization. Support for multiprocessor systems was also added to UTIME.

The RTSS also grants users the ability to change real-time schedules on the fly, rather than having to wait for the last submitted schedule to complete before the next can be used. The RTSS generates cyclic schedules with periods of 10ms, 20ms, 40ms, or 80ms. This period depends upon the total CPU-usage request from all agents on a single machine. If the total request is from 0-10% of the CPU, the schedule will have a period of 10ms. If the request is from 11-20%, the schedule will be 20ms long. A request from 21-40% generates a 40ms schedule and a request from 41-66% (66% is the maximum total percentage request possible) generates a schedule of 80ms. These particular schedule lengths are used in order to correctly calculate the actual cpu-usage for each thread of an agent (a thread may request 10%, but may only use 5% due to blocking or sleeping). The CPU-usage calculation code runs every 80ms and determines each registered agent thread's actual use of the CPU during the last calculation period. Since the greatest common multiple of each possible cyclic schedule length is 80ms, this calculation will be accurate.

There are some cases for agents where requesting a percentage of the CPU for a single thread is difficult to determine because that particular thread's progress is dependent upon the progress of other threads. In these cases, requesting a CPU percentage for a group of threads is more beneficial. KURT now provides a group-aware scheduler for this purpose. When a group scheduling timer event occurs, the top-level KURT scheduler determines which group the event was for and then calls the group scheduling routine. This routine examines each thread in the group and grants context to the first non-blocked thread. This is a very basic policy. We are evaluating the use of more complicated group scheduling policies.

4.2. The KU KickStart Tools (KUKT)

The KU KickStart Tools (KUKT, pronounced "cooked") are extensions to the standard Red Hard Kickstart utility that significantly automate Linux system software installation and configuration, specifically for support of experimentation. The existing Red Had utility is intended to be interactive with the user as they install software on their Linux system, typically from CD. However, several projects use sets of Linux machines for experimental studies in

which reproducibility of the system software and configuration is crucial. The KUKT extensions to Red Hat's KickStart utility have been designed to automate the installation process, and to emphasize reproducibility of system configuration in service of good experimental technique.

The KUKT system is divided into two parts: client and server. The client portion addresses how a client machine can have a specific configuration of Linux installed upon it. This includes utilities to create boot floppies, as well as procedures for supporting remotely controlled initiation of system installation. The server portion is the bulk of the KUKT approach, and controls every aspect of system software installation on the target machines.

KUKT assumes that a server machine, the KUKT server, is present on an Ethernet network shared with the experimental machines being controlled. The KUKT approach is based on a hardware and software profile for each machine being configured. The hardware profile provides information required to configure the kernel required to support system configuration. One of the most important elements of the hardware profile is, for example, the type of Ethernet card in the machine so that the client machine being configured can communicate with the KUKT server during configuration. The software profile specifies all software that is installed on the machine. This includes all Linux system software, of course, but also all user generated software used for the experiment.

The KUKT server provides storage for all profile files required to manage the set of machines under its control, as well as for the software whose installation is described by the profiles. Several groups of investigators can thus share a set of machines by writing appropriate profiles for each machine in their experiment, and by loading the software unique to their experiments on the machine. Changing use of a set of experimental machines from one group to another is then as simple as instructing the machines to install the configuration described by the software profiles provided by the group taking over use of the machines.

While the KUKT tools already existed for KU internal use, we had to perform a significant amount of work to make them ready for more general use. We have produced client and server software that should work in a generic experimental environment, and the documentation required by those wishing to set it up. While one or two refinements may be desired, we believe the delivered software is complete, and will require no further development unless users encounter problems.

4.3. Data Streams

The Data Streams (DS) uses a high resolution timer, the CPU time stamp counter on Intel Pentium class processors, to place all events in the system on the same time line. This enables the user to relate events in different processes to each other, and to events at the system level. Under the ANTS project we implemented the DS approach in the Linux kernel (DSKI) and with user-level interfaces (DSUI) Java. It provides counters, events, and histograms as native object types for data collection purposes. The Java-DSUI has been designed to be lightweight; it should have minimal impact on the programs it monitors, and is suitable for both debugging and performance measuring purposes.

The two most fundamental ideas in the data streams approach to performance evaluation are the "instrumentation point " and the "data source". Instrumentation points are placed in the code for

which performance information is desired, at places where specific data sources should be updated. The term "data source" is used from the perspective of a process gathering data. In that sense, a data source is thus any separate element provided by the DS from which an interested user program can obtain performance information. A "data stream" refers to the set of data gathered by a particular user process from one or more data sources in the course of an experiment. The word "stream" is appropriate here since the data gathered is time-stamped, and can thus be placed in a total order. The data gathering process configures a data stream by associating a set of data sources with each data stream as it desires.

Implementing the data streams approach for user level processes we use a specification file to describe the set of data sources that should be used during a given experiment and supply the name of the specification file to the process during invocation. In Java we use a specification file to describe the name space, and a utility program to generate a Java class specification in a given package describing the data source name space of the Java program. This approach reduces the labor of instrumenting Java code to a reasonable level, while providing the experimenter with maximum control. New data sources can be defined and used with minimal effort, and the exact set of data sources required for an experiment can be specified.

The Java version of the DSUI takes advantage of its object orientation to direct the output of each data source to the desired output destination using the generic print method and the existing Java output stream facility. The associated streams can be directed to files, memory buffers, devices, etc.

The Java version of the DSUI provides powerful and flexible support for instrumenting application code, and for collecting performance evaluation information during experiments. Users can create sets of data sources, define the name space of data sources, and use the defined name space to control data source use. Since the DSUI uses the Pentium TSC to gather the time stamps, event streams from different sources on the same machine can be merged to represent all events on a common time line. The availability of counters and histograms as well as generic events lowers the overhead of gathering certain common types of data. The Java DSUI has passed both static and dynamic tests. The static tests were performed by ESC/Java (<URL:http://research.compaq.com/SRC/esc/>). The dynamic tests were written and applied in an *ad hoc* manner.

4.4. The Communication Server

The Communication Server is a replacement for the slow RF links. It is intended to provide a simulation of the RF links, but allow the latencies and drop rates to be controlled. The Communication server supports the narrow and wide formats of the RF channels, has message logging for debugging and accounting purposes, and supports collision and bandwidth modeling.

The desired bandwidth can be specified on the command line, using the -b option. If no bandwidth is specified, the default is 19600 bits per second. When collision modeling is enabled, the structure of the message handling is changed in the following manner:

      a. When the client handler thread receives a message from the client, it calculates the latency for the message according to the modeled bandwidth.

      b. Once a message is received by the client handler thread, the xmit_start_time field of the message structure is assigned the current timestamp value, indicating the

start time of the message. The time it would take to transmit the message on the modeled channel is calculated to be delay = (message_size * 8) / bandwidth. The xmit_end_time of the message structure is then assigned the value xmit_start_time + delay.

    c.   The client handler thread inserts the message in the channel list. Before it stores the message in the channel list, it checks for any collisions with existing messages on that channel. This is done by comparing the start time of the new message with the end times of the existing messages. If the start time is less than the end time of any message, both messages are marked as collided.

    d.   The output control thread is then signaled, indicating that a new message has been put in the channel list.

    e.   When the output control checks for messages in the channel list, it checks for messages whose xmit_end_time has expired. If such a message is marked as collided, it is simply discarded. Otherwise, the message is delivered to the clients receiving on the appropriate channel.

    f.   Unexpired messages (i.e., those with xmit_end_time in the future), cannot be delivered. To avoid wasting CPU cycles, the output control thread sleeps when there are no expired messages.

We tested the Communication Server extensively to identify its performance. We tested, among others, message integrity, message ordering, and throughput performance using C-clients, Java clients and the CP code. We also tested the collision model without network delays and then with the actual network and identified the delays associated with it.

4.5. Modifications to the CP Code

We were charged with discovering the causes of unacceptably high message loss rates when using the RF hardware, and with removing or alleviating those causes as possible. During our work we discovered and compensated for several contributing factors.

1. We discovered possible interference between the sending and receiving threads in the CP code. We inserted explicit synchronization between those threads, and saw another large drop in message loss rates as a result. Our analysis shows that a running receive thread interferes with the transmission of messages. Our solution causes the receive thread to block while the sending thread transmits.

2. We discovered that a bug in the Linux 2.2 series of kernels affects the message loss rate. This bug, in the serial driver, causes a process using a serial line to be suspended for long periods of time, occasionally. These long suspensions resulted in missing some of the bytes of a message, and consequently the entire message had to be discarded. The Linux 2.4 series of kernels do not have this bug, so upgrading the kernel is sufficient to improve the message loss rate. For those unable to upgrade, we provided a workaround. This workaround schedules the CP code process as a real-time process, which avoids the incorrect behavior. However, it requires that the CP code be executed by a privileged user.

3. We improved the `makefile` of the CP code so that it is no longer necessary to do a `'make configure'` before a `'make'` when compiling. The `makefile` determines if a `'make configure'` is required and does it automatically.

4. We edited the CP code so that it is no longer necessary to manually comment out or uncomment the '#include ' in the file `LynxHPDriver.c`. The make files and `LynxHPDriver.c` have been modified to automatically determine if the code is being compiled on a LinuxOS or not and then does the right thing.

5. If the Linux kernel version is less than or equal to 2.3.0 the `LynxHPDriver.c` file is automatically changed during compilation (using `#ifdefs`) to use RoundRobin process scheduling during message transmissions to work around the problem in the serial driver that causes the '`tcdrain`' function to take too long to return, causing transmission overlaps during the PingPong test. If the kernel version is newer than 2.3.0 then message transmissions occur during 'normal' process scheduling since the newer kernels do not have the serial driver problem.

6. Modified the LynxHP, TCP and UDP communication drivers to set a local time stamp when a message has completed transmission. The MTIRadar driver was modified to use the message transmission time stamp to determine the minimum amount of time to wait before it is safe to start a radar measurement to avoid interference from the LynxHP transmitter. This used to be a fixed 50msec wait (CP 1.4). The wait is now calculated dynamically to be between 0 and 180msec. Tests were preformed to determine the minimum amount of time to wait after a LynxRF message has finished transmission before it is safe to start a radar measurement. This was determined to be 180mS. When a radar measurement is requested, the MTI radar driver determines the difference between the current time and the last message transmission completion time. If the difference is greater than 180msec the radar measurement starts immediately, otherwise there is a wait of 180msec- (currentTime - TxFinishTime).

7. The knowledge of the structure of the '`msgSent`' and '`msgReceived`' time stamps for the `control/Message` class was spread across the `control/MessageQueueServiceThread`, `control/Message`, `control/Measurement`, `control/Response` and `control/UserMessage` classes. This code was moved where is should be in the `control/Message` class. This significantly improves code maintenance if there are any future changes in the `control/Message` fields.

8. Added code to automatically detect and warn the user, at run time, if the serial cables for the LynxHP or MTIRadar are not connected to the host.

9. The `control/UserMessage` class is now a descendent of the `runtime/Message` class. The consequence of this is that one less class instance is created for each user message received and several byte arrays do not have to be created and copied. When running on our test machines in native threads there is an increase in message rate of the PingPong test from 13.6 msgs/sec to 15.3 msgs/sec. When running in green threads there is no measurable speed difference. The speed is 12.87 msgs/sec

Altogether, our code improvements have resulted in message loss rates of substantially less than 1% for all of our systems, regardless of message length, CPU load, or age of the system.

5. Leave-Behinds and Publications
We delivered the following code modules to DARPA:
- Agent Code - includes profiler, task manager, negotiation manager, reasoner, etc.
- Radar Calibration Fix - our code to correct the operational problem (sector switching) of the radar calibration code

- o Case-Based Reasoning code
- o 3D Visualization - 3D real-time visualization and scene input parser
- o Radar Sensor-Sector Target Computation
- o Genetic Algorithm Module for Case Generation
- o CP Performance Enhancements
- o KU Real Time Linux (KURT)
- o KU Kickstart Tools (KUKT)
- o Communication Server
- o Instrumentation Java DSUI, instrumentation tools for Java programs

We produced the following publications as a direct result of the project:

1. Soh, L-K, C. Tsatsoulis, and H. Sevay. 2003. "A Satisficing, Negotiated, and Learning Coalition Formation Architecture," in: *Distributed Sensor Networks: A multiagent perspective*, C. Ortiz, V. Lesser and M. Tambe (Eds.), Kluwer, (to appear).
2. Soh, L-K. and C. Tsatsoulis. 2002. "Satisficing Coalition Formation among Agents," *1ˢᵗ Int. Conf. On Autonomous Agents and Multiagent Systems*, 1062-63.
3. Soh, L-K. and C. Tsatsoulis. 2002. "Allocation Algorithms in Dynamic Negotiation-Based Coalition Formation," *Workshop on Teamwork and Coalition Formation* (held during the *1ˢᵗ Int. Conf. On Autonomous Agents and Multiagent Systems*), 16-23.
4. Soh, L-K. and C. Tsatsoulis. 2002. "Learning to Form Negotiation Coalitions in a Multiagent System," *AAAI Spring Symposium on Collaborative Learning Agents,* 106-12.
5. Soh, L-K. and C. Tsatsoulis. 2001. "Agent-Based Argumentative Negotiations with Case-Based Reasoning," *AAAI Fall Symposium on Negotiation Methods for Autonomous Cooperative Systems,* 16-25.
6. Soh, L-K., C. Tsatsoulis, M. Jones and A. Agah. 2001. "Evolving Cases for Case-Based Reasoning Multiagent Negotiations," in: *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO-2001, ,* San Francisco, CA: Morgan Kaufmann Publishers, 909.
7. Soh, L-K. and C. Tsatsoulis. 2001. "Combining Genetic Algorithms and Case-Based Reasoning for Genetic Learning of a Casebase: A Conceptual Framework," in: *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO-2001, ,* San Francisco, CA: Morgan Kaufmann Publishers, 376-383.
8. Niehaus, D., C. Tsatsoulis, W. Dinkel, and A. Gautam. 2001. "An Infrastructure for Real-Time, Reflective Intelligent Agents," *Ninth International Workshop on Parallel and Distributed Real-Time Systems and Sixth International Workshop on Embedded/ Distributed HPC Systems Applications,* San Francisco, CA.
9. Soh, L-K. and C. Tsatsoulis. 2001. "Reflective Negotiating Agents for Real-Time Multisensor Target Tracking," *Int. J. Conf. On Artificial Intelligence (IJCAI-01),* Seattle, WA, 1121-1127.
10. Dinkel, W., A. Gautan and D. Niehaus. 2000. *Comparison of Linux/RK and KURT for Use in ANTS,* ITTC Technical Report, The University of Kansas.

6. References

Allen, J. F. (1983). Maintaining knowledge about temporal intervals, *Communications of the ACM*, *26*(11), 832-843.

Allen, J. F. (1991). Time and time again: the many ways to represent time, *International Journal of Intelligent Systems*, *6*(4), 341-355.

Allen, J. F. and Ferguson, G. (1994). Actions and events in interval temporal logic, *Journal of Logic and Computation, Special Issue on Actions and Processes*, *4*(5), 531-579.

Brazier, F. and Treur, J. (1996). Compositional modelling of reflective agents. In *Proceedings of the 10th Knowledge Acquisition for Knowledge-Based Systems Workshop (KAW'96),* Banff, Alberta, Canada, 13/1-13/12.

Cohen, P. R. and Levesque, H. J. (1990). Intention is choice with commitment, *Artificial Intelligence*, *42*(2-3), 213-261.

Faratin, P., Sierra, C., and Jennings, N. R. (1998). Negotiation decision functions for autonomous agents, *International Journal of Robotics and Autonomous Systems*, *24*(3-4), 159-182.

Galliers, J. R. (1998). A strategic framework for multi-agent cooperative dialogue, in *Proceedings of ECAI'88,* Munich, Germany, 415-420.

Kolodner, J. (1993). *Case-Based Reasoning*. Morgan Kaufmann, San Mateo, CA.

Lander, S. and Lesser, V. (1992). Customizing distributed search among agents with heterogeneous knowledge, in *Proceedings of CIKM-92*, Baltimore, MD, 335-344.

Noriega, P. and Sierra, C. (1996). Towards layered dialogical agents, in *Proceedings of ECAI Workshop on ATAL'96*, Budapest, Hungary, 157-171.

Parsons, S., Sierra, C., and Jennings, N. R. (1998). Agents that reason and negotiate by arguing, *Journal of Logic and Computation, 8*(3), 261-292.

Rao, A. and Georgeff, M. (1991). Modeling rational agents within a BDI-architecture, in *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning*, 473-484.

Rao, A. and Georgeff, M. (1995). BDI agents: from theory to practice, in *Proceedings of ICMAS-95*, San Francisco, CA, 312-319.

Srinivasan, B., Pather, S., Hill, R., Ansari, F., and Niehaus, D. (1998). A firm real-time system implementation using commercial off-the shelf hardware and free software, in *Proceedings of RTAS-98,* June Denver, CO, 112-119.

Tambe, M. (1997). Towards flexible teamwork, *Journal of Artificial Intelligence Research*, *7*, 83-124.

Vere, S. A. (1983). Planning in time: windows and durations for activities and goals, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *5*(3), 246-267.

Wooldridge, M. and Jennings, N. (1995). Intelligent agents: theory and practice, *The Knowledge Engineering Review,* 10(2), 114-152.

Zlotkin, G. and Rosenschein, J. S. (1996). Mechanism design for automated negotiation, and its application to task oriented domains, *Artificial Intelligence*, *86*(2), 195-244.